CRANFIELD UNIVERSITY

MICHAŁ CZAPIŃSKI

Solvers on Advanced Parallel Architectures with Application to Partial Differential Equations and Discrete Optimisation

School of Engineering
Energy & Power Engineering Division

PhD Thesis
Academic year: 2013–2014

Supervisor: Prof. Chris P. Thompson
May 2014

CRANFIELD UNIVERSITY

School of Engineering
Energy & Power Engineering Division

PhD Thesis
Academic year: 2013–2014

Michał Czapiński

Solvers on Advanced Parallel Architectures with Application to Partial Differential Equations and Discrete Optimisation

Supervisor: Prof. Chris P. Thompson
May 2014

This thesis is submitted in partial fulfilment of the requirements for the degree of *Philosophiæ Doctor* (PhD)

To my parents and grandparents,
to whom I owe so much.

Abstract

This thesis investigates techniques for the solution of partial differential equations (PDE) on advanced parallel architectures comprising central processing units (CPU) and graphics processing units (GPU). Many physical phenomena studied by scientists and engineers are modelled with PDEs, and these are often computationally expensive to solve. This is one of the main drivers of large-scale computing development.

There are many well-established PDE solvers, however they are often inherently sequential. In consequence, there is a need to redesign the existing algorithms, and to develop new methods optimised for advanced parallel architectures. This task is challenging due to the need to identify and exploit opportunities for parallelism, and to deal with communication overheads. Moreover, a wide range of parallel platforms are available — interoperability issues arise if these are employed to work together.

This thesis offers several contributions. First, performance characteristics of hybrid CPU-GPU platforms are analysed in detail in three case studies. Secondly, an optimised GPU implementation of the Preconditioned Conjugate Gradients (PCG) solver is presented. Thirdly, a multi-GPU iterative solver was developed — the Distributed Block Direct Solver (DBDS). Finally, and perhaps the most significant contribution, is the innovative streaming processing for FFT-based Poisson solvers.

Each of these contributions offers significant insight into the application of advanced parallel systems in scientific computing. The techniques introduced in the case studies allow us to hide most of the communication overhead on hybrid CPU-GPU platforms. The proposed PCG implementation achieves 50–68% of the theoretical GPU peak performance, and it is more than 50% faster than the state-of-the-art solution (CUSP library). DBDS follows the Block Relaxation scheme to find the solution of linear systems on hybrid CPU-GPU platforms. The convergence of DBDS has been analysed and a procedure to compute a high-quality upper bound is derived.

Thanks to the novel streaming processing technique, our FFT-based Poisson solvers are the first to handle problems larger than the GPU memory, and to enable multi-GPU processing with a linear speed-up. This is a significant improvement over the existing methods, which are designed to run on a single GPU, and are limited by the device memory size. Our algorithm needs only 6.9 seconds to solve a 2D Poisson problem with 2.4 billion variables (9 GB) on two Tesla C2050 GPUs (3 GB memory).

Acknowledgements

It is my privilege to express gratitude to people who have helped and supported me during work on this thesis. Foremost, I would like to acknowledge the work of my supervisor Professor Chris Thompson: his invaluable advice, constant support, and many inspiring conversations. I would also like to thank Doctor Robert Sawko and Doctor Stuart Barnes for their countless comments improving my work, proof-reading my papers and reports, and that they have always found time for me.

Special thanks go to Doctor Mark Stillwell who volunteered to assist me during the final year of my studies. His experience and vast knowledge helped me to broaden my perspective and improve my skills in many areas. In his support, Doctor Stillwell has gone far beyond the call of duty, and practically co-supervised the thesis.

During my studies at Cranfield University, I was a member of the Handball Club and the Walking Society where I have met many wonderful people. Sports and hiking weekends away were crucial to keep the right balance between work and social life.

I would like to express my great appreciation to all my dearest friends who supported, and believed in me, and who have always stood by my side. The full list would be too long to fit in this page, however I would like to name a few closest to my heart: Robert Sawko (again!), Paweł Jaworski, Michał Orłowski, Artur Kopczyński, Bernadetta Brzęczek, Sarah Bergin, and Milena and Marcin Traczyk. The warmest thanks and love go to very special British friends Pat and Bill Davies, and Sue and Paul Lowe, who made living in England feel like back at home.

Last, but definitely not least, my infinite gratitude and love goes to my family, especially to my beloved wife Gosia, and to my parents Joanna and Jacek, for their love, patience, and everlasting support. I would never be able to accomplish so much without their help.

Contents

List of Tables

List of Figures

List of Algorithms

Glossary

Abbreviations

API	Application programming interface
AXPY	AXPY operation on vectors: $\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$
CG	Conjugate Gradients
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
DBDS	Distributed Block Direct Solver
DCG	Distributive Conjugate Gradients
DFPS	Distributed Fast Poisson Solver
DFT	Discrete Fourier Transform
DST	Discrete Sine Transform
FDM	Finite Difference Method
FEM	Finite Element Method
FFT	Fast Fourier Transform
FPS	Fast (FFT-based) Poisson Solver
FVM	Finite Volume Method
GEMM	Dense matrix-matrix multiplication operation
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics processing unit
HPC	High-Performance Computing
HPSL	High-performance Parallel Solvers Library
MPI	Message Passing Interface
PCG	Preconditioned Conjugate Gradients

PDE	Partial differential equation
PE	Processing element (a CPU core or a GPU)
SOR	Successive Over-Relaxation
SPD	Symmetric and positive-definite (matrix)
SpMV	Sparse matrix-vector multiplication

Notation

$\alpha, \ldots, \omega$	Scalars
$\mathbf{a}, \ldots, \mathbf{z}$	Vectors
$\mathbf{A}, \ldots, \mathbf{Z}$	Matrices
$\alpha^{(r)}$, $\mathbf{a}^{(r)}$, $\mathbf{A}^{(r)}$	Scalar/vector/matrix in r-th iteration
$\mathbf{A}^T$	Transpose of matrix $\mathbf{A}$
$\mathbf{I}$	Identity matrix
$\mathbf{x}^*$	Exact solution to a linear system
i, j, k	Indices for elements in vectors, matrices, and grids
n, m, l	Problem size
P	The number of subdomains (or processing elements)
R	Step number in an iterative method
a_i, $A_{i,j}$	Vector/matrix/grid element
$\mathbf{a}_i$, $\mathbf{A}_{i,j}$	Vector/matrix/grid block
$diag(a, b, c)$	Tridiagonal matrix with a on lower diagonal, b on main diagonal, and c on upper diagonal
$\kappa(\mathbf{A})$	Condition number of matrix $\mathbf{A}$
ρ	Convergence rate of iterative method
$\rho(\mathbf{A})$	Spectral radius of matrix $\mathbf{A}$
ε	Machine floating-point representation error
u_t, u_{xy}	Notation for partial derivatives: $u_t := \frac{\partial u}{\partial t}$, $u_{xy} := \frac{\partial^2 u}{\partial x \partial y}$
$\| \cdot \|$	Vector/matrix norm
$(\mathbf{a}, \mathbf{b})$	Vector dot (inner) product
$\mathcal{O}(\cdot)$	"Big O" asymptotic notation

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."

Steve Jobs (2008)

"Number of processor cores: 448."

Tesla C2050 specifications (2010)

"There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are."

Gary R. Montry

"Life isn't mostly symmetric positive definite."

John Shadid

Chapter 1

Introduction

Many physical phenomena in science and engineering are modelled with partial differential equations (PDEs). The list of possible applications includes computational fluid dynamics, weather forecasting, quantum mechanics, astrophysics, geophysics, seismic analysis, electrostatics, electromagnetism, and mechanical engineering.

Due to the complexity of real-life scenarios, the application of analytical methods is often impractical, as the solutions are difficult, or even impossible, to find. To overcome this limitation, a *discrete* representation is used instead. A continuous formulation is discretised, and the problem is transformed to finding the solution of a system of algebraic equations. However, even using a discrete representation many physical systems require an enormous amount of computation. This became one of the main drivers for rapid development in large-scale computing.

The performance of modern central processing unit (CPU) cores is close to physical limits — further improvement is being achieved by putting several cores on a single chip (Schauer, 2008). In contrast, graphics processing units (GPUs) typically consist of hundreds of cores, which are much simpler than sophisticated CPU cores. GPUs were initially developed to meet the insatiable market demand for real-time, high-definition 3D graphics in computer games, however they also provide an attractive platform for general-purpose computation (GPGPU). In consequence, parallel and distributed processing is the future of large-scale computing.

There are many well-established PDE solvers (e.g. based on Conjugate Gradients or Multigrid Methods), but they are often inherently sequential and thus cannot

take full advantage of modern, massively parallel hardware. This creates a need to redesign the existing algorithms, and to develop completely new methods optimised for today's advanced parallel architectures. This task is challenging due to the need to identify and exploit opportunities for parallelism, and to deal with communication overheads. Furthermore, a large number of parallel platforms are available (e.g. CPU clusters, GPUs, Field-Programmable Gate Arrays), varying significantly in performance and applicability to different classes of problems (e.g. CPU clusters are better suited for parallel tasks that require relatively infrequent communication; GPUs are ideal for compute-intensive problems). If these platforms are employed to work together, then interoperability issues arise and have to be addressed. A wide range of parallel systems is considered in this project: small- and large-scale multi-core CPU clusters with Ethernet and Infiniband interconnect, various GPU models with different features and capabilities, and finally, hybrid CPU-GPU installations.

1.1 Contributions

The research presented in this thesis is primarily focused on techniques for the solution of partial differential equations on advanced parallel architectures. The project offers several contributions in the challenging field of designing massively-parallel algorithms that can take advantage of huge computational capabilities of high-performance computing systems comprising thousands of CPU and GPU cores.

The first contribution is a detailed analysis of performance characteristics of hybrid CPU-GPU platforms. Possible bottlenecks and optimisation techniques to address them are discussed in three case studies. The proposed methods to reduce the communication overhead, and for load balancing in heterogeneous distributed memory multi-GPU systems allowed us to implement the Hybrid Jacobi Solver for the 2D Laplace's Equation. This case study was extended to create the Hybrid SOR Solver for the 2D Poisson's Equation, where efficient handling of non-trivial data dependencies and optimisation of in-memory data layout was addressed. The third case study — the Multistart Tabu Search for the Quadratic Assignment Problem — shows how to implement more complex algorithms and data structures on GPUs efficiently.

The second contribution is the Distributed Block Direct Solver (DBDS) following the Block Relaxation scheme (Young, 1971, Ch. 14). DBDS is a novel multi-GPU

implementation motivated by the previously published Distributive Conjugate Gradients (DCG; Becker, 2006; Becker and Thompson, 2006) method for the solution of large, sparse linear systems. In comparison to the DCG algorithm, DBDS is more robust and allows us to solve a wider range of linear systems. In addition, the DBDS method is customisable and can be easily optimised for a particular problem. Moreover, an analysis of the theoretical properties is provided: the class of systems for which the DBDS method converges is identified, and the upper bound for the convergence rate is derived. The techniques that improve the performance of the basic DBDS algorithm are proposed: subdomain overlapping, and mixed-precision iterative refinement.

The final contribution of this thesis is a range of highly scalable and optimised multi-GPU implementations of 2D and 3D Poisson solvers based on Fourier analysis. Two novel approaches were considered. The first aims at reducing the communication cost in distributed memory systems. In consequence, the scalability is significantly improved — a linear speed-up was observed on 128 CPU cores, and a performance of 300 GFLOPS was achieved. The second class of implementations was optimised for shared memory systems. In this area, a few GPGPU solutions have been published already, but their scalability is significantly limited.

Thanks to the innovative streaming processing technique proposed in this thesis, our FFT-based Poisson solvers are the first to handle problems larger than the GPU memory, and to enable multi-GPU processing with a linear speed-up. This is a significant improvement over the existing methods, which are designed to run on a single device, and the maximum Poisson problem size is limited by the relatively small GPU memory. Our method is capable of solving problems with billions of variables in a matter of seconds: a 2D Poisson grid with 2.4 billion variables (9 GB in single precision) can be solved in 6.9 seconds on two Tesla C2050 GPUs (3 GB memory). A 3D Poisson grid with 2.1 billion variables (8 GB in single precision) can be solved in 10.8 seconds on the same installation.

1.2 Thesis Outline

The thesis is organised as follows. Chapter 2 presents the review of literature on solution methods for partial differential equations, and parallel computing in dis-

tributed and shared memory environments. The remainder of this dissertation is divided into three parts.

Part I contains three case studies providing a detailed analysis of performance characteristics of hybrid CPU-GPU platforms. Chapter 3 presents the design and development of the Hybrid Jacobi Solver for 2D Laplace's Equation. Then, Chapter 4 discusses techniques used to handle non-trivial data dependencies that arise in the Hybrid SOR Solver for the 2D Poisson's Equation. Finally, Chapter 5 presents an application of the GPUs in Operations Research, using the Multistart Tabu Search for the Quadratic Assignment Problem, and shows how to implement complex algorithms and data structures on the GPU.

Part II introduces the Distributed Block Direct Solver for the solution of large, sparse linear systems on distributed memory parallel platforms. Chapter 6 presents a highly optimised GPU implementation of the Preconditioned Conjugate Gradients with a range of preconditioners: diagonal, polynomial, and based on incomplete Cholesky and LU factorisations. The techniques discussed there, are later used in the novel hybrid CPU-GPU implementation of the Distributed Block Direct Solver method, which is derived and analysed in Chapter 7. The results of numerical experiments on hybrid DCG and DBDS implementations are discussed in Chapter 8.

Finally, Part III presents a range of Fast Poisson Solvers based on Fourier analysis. The first one is designed for distributed memory hybrid CPU-GPU platforms and it is described in Chapter 9. Then, the algorithm for the 2D Poisson problem designed for shared memory multi-GPU systems, and the streaming processing technique are presented in Chapter 10. The FFT-based solver for the 3D Poisson problem introduces new performance issues and is addressed separately in Chapter 11.

The conclusions from this project, and the possible directions of further research and development are summarised in Chapter 12. Appendices provide an overview of the CUDA platform for GPGPU (Appendix A), present specifications of CPUs and GPUs (Appendix B), and the numerical problems (Appendix C) used in experiments, and finally, list the publications and conference talks prepared during this project (Appendix D).

Chapter 2

Literature Review

2.1 Partial Differential Equations

The history of partial differential equations started in the eighteenth century, when d'Alembert formulated an equation to model string vibration. The research was then continued by Euler (uniqueness of the Wave Equation solution), Fourier (Fourier series), Cauchy (initial value problem), Laplace and Poisson (Laplace's and Poisson's Equations). For the detailed history of the theory of partial differential equations refer to Brezis and Browder (1998).

2.1.1 Types of Partial Differential Equations

An overview of the most common PDEs with their applications is provided below.

Linear Advection Equation

$$u_t + a \cdot u_x = 0\,, \tag{2.1}$$

where $a < 0$ is a constant coefficient describing the *propagation speed*. It is easy to show that

$$u(x,t) = u_0(x - a \cdot [t - t_0]), \tag{2.2}$$

where $u_0(x)$ is the value at moment t_0, is the solution to the *initial value problem*. This makes Equation (2.1) one of the simplest equations that describes

the propagation of a scalar quantity in a spatial domain. The Linear Advection Equation is used to model transport of some property within a domain, e.g. heat, humidity, or salinity in the medium.

Despite its simplicity, the equation proved to be considerably difficult to solve in a discretised domain. Dispersive and dissipative errors ensure that there is no universal scheme which deals with all possible initial conditions. The existence of the exact analytical solution, makes the Linear Advection Equation a good choice for benchmarking purposes (Leonard, 1991).

Non-linear Advection Equation (Burgers' Equation)

$$u_t + u \cdot u_x = \nu \cdot u_{xx} \,, \tag{2.3}$$

introduces non-constant propagation speed and captures the behaviour of fluids. If *viscosity coefficient* $\nu = 0$, Equation (2.3) becomes the Inviscid Burgers' Equation:

$$u_t + u \cdot u_x = 0 \,, \tag{2.4}$$

for which the solution can develop *discontinuities* (shock formations). This makes Equation (2.4) exceedingly challenging for numerical methods.

Heat Equation

$$u_t = \sigma \cdot \nabla^2 u \,, \tag{2.5}$$

$$\text{where} \quad \nabla^2 u = \sum_{i=1}^{n} \left(\frac{\partial^2 u}{\partial x_i^2} \right) , \tag{2.6}$$

and σ is the *diffusion coefficient*, is the simplest *parabolic equation*. Patankar (1980) used Equation (2.5) as a model problem for Finite Difference and Finite Volume Methods. The main application of the heat equation is to model heat distribution over time, but it is also useful in financial mathematics to solve the Black-Scholes Equation (European-style options pricing; Wilmott et al., 1995), or in image analysis (Perona and Malik, 1990).

Second Order Wave Equation

$$u_{tt} - a^2 \cdot u_{xx} = 0 \,, \tag{2.7}$$

where a describes the speed of wave propagation, is the simplest example of a *hyperbolic equation*. In contrast to the linear advection, here disturbances travel both ways. Equation (2.7) was initially used to model string vibration, but it is applicable to any one- or more dimensional *wave-guides*.

Poisson's Equation

$$\nabla^2 u = f\,, \tag{2.8}$$

where ∇^2 is defined as in Equation (2.6), is certainly one of the most popular PDEs. The governing equations of incompressible fluid dynamics can be recast to this form (Pope, 2000). Equations of steady, inviscid, and irrotational fluid simplify to the special case of Poisson's Equation — *Laplace's Equation*, where $f \equiv 0$. Laplace's Equation can be also used to describe magnetic and electric systems. Equation (2.8) is the simplest example of an *elliptic equation*.

Navier-Stokes Equations describe the motion of fluid substances. The equations are derived by applying Newton's Second Law to fluid motion, and assuming that the stress in the fluid is the sum of pressure and diffusing viscous terms. For more information refer to the book by Temam (2001).

In this thesis, Poisson's Equation is considered as a model problem since it emerges in many applications. Furthermore, algorithms finding a solution to Poisson's Equation typically experience low computation-to-communication ratio, which makes the effective parallelisation particularly challenging.

2.1.2 Discrete Representation of Partial Differential Equations

The discretisation of PDEs to finite numerical representation typically consists of two steps: *domain* and *equation* discretisations, which are described in the following subsections. There are also techniques called *mesh-free methods*, which do not require the underlying mesh, but they are out of the scope of this project. For more information on mesh-free methods refer to Fries and Matthies (2003), Idelsohn et al. (2003), Ferreira et al. (2008), and Griebel and Schweitzer (2010).

Domain Discretisation Techniques

At this stage, a *finite* number of points are selected from the physical, continuous space, and form a *mesh*. The values of variables involved in the numerical simulation are only computed at mesh points. The accuracy of the numerical approximation directly depends on the size (number of points), and the shape of the mesh.

Meshes can be classified based on the way the points are organised. The mesh is *structured*, if the organisation of its points follow some general rule[1], otherwise it is *unstructured*. Due to the structured nature of grids, the algorithms based on them are much easier to design and implement. However, grids are difficult to define when applied to complex geometries due to lack of flexibility. Unstructured meshes fix that shortcoming, and can provide the same accuracy as structured grids, requiring fewer points. A compromise between these two approaches are *block-structured* meshes, which divide the physical space into several contiguous subdomains, and a separate structured grid is generated for each subdomain. The well-defined mesh structure is maintained, however interaction between adjacent blocks is more complicated. Block-structured meshes work well when applied to complex geometries.

For details on mesh generation methods refer to Knupp and Steinberg (1994) and Liseĭkin (1999) for structured grids, Huang (1997) and Chand (2005) for block-structured meshes, and Löhner et al. (2001) for unstructured meshes. Furthermore, Chrisochoides (2005) provides a survey of parallel mesh generation techniques.

Equation Discretisation Techniques

Once a mesh is generated, equation discretisation can be performed. The dependencies between variables values at the mesh points, coming from the system of PDEs, have to be expressed in a finite number of algebraic operations, typically linear equations. The method used to accomplish that, has a direct impact on the accuracy of the numerical model, and on the complexity of the equations.

[1] Structured meshes are often referred to as *grids*.

The most commonly used methods are *Finite Difference Method* (Orkisz, 1998), *Finite Volume Method* (Eymard et al., 2000; LeVeque, 2002; Versteeg and Malalasekera, 2007), and *Finite Element Method* (Reddy, 1984; Pepper and Heinrich, 1992; Akin, 1994; Zienkiewicz et al., 2005).

2.2 Systems of Linear Equations

There are many methods for solving systems of linear equations in the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,. \tag{2.9}$$

These methods can be divided into *direct methods*, and *indirect* or *iterative methods*, and are in the main scope of this project. An overview of early iterative methods can be found in Varga (1999). The more recent surveys are provided by Barrett et al. (1994) and Saad and van der Vorst (2000).

An overview of direct methods is given in Subsection 2.2.1, followed by presentation of the simplest, stationary iterative methods in Subsection 2.2.2. More sophisticated gradient methods are described in Subsection 2.2.3. Other iterative methods are summarised in Subsection 2.2.4. Finally, Subsection 2.2.5 provides a brief overview of preconditioning techniques used to improve the performance of iterative methods.

2.2.1 Direct Methods

Gaussian elimination is among the most popular methods for solving systems of linear equations. It was invented independently in ancient China, and later in Europe in the works of Isaac Newton. As a result of confusion over the history of the subject, it was named after Carl Friedrich Gauss, who devised a notation for symmetric elimination (Grcar, 2011).

For an $n \times n$ matrix $\mathbf{A}$, solving a system of linear equations with Gaussian elimination requires approximately $2n^3/3 = \mathcal{O}(n^3)$ floating-point operations, however the intermediate entries can grow exponentially (Fang and Havas, 1997). This method

is numerically stable for diagonally dominant[2] or positive-definite matrices[3]. In the case of general matrices, *full pivoting* must be used to ensure numerical stability. In practice, *partial pivoting* is enough to consider Gaussian elimination numerically stable, however Golub and Van Loan (1996, §3.4.6) show examples when Gaussian elimination with partial pivoting is unstable.

LU factorisation is another direct method for solving linear systems. Here, matrix $\mathbf{A}$ is first decomposed into two matrices: $\mathbf{L}$ (lower-triangular) and $\mathbf{U}$ (upper-triangular), that satisfy $\mathbf{A} = \mathbf{LU}$. Since the LU factorisation is not unique, additional constraints are imposed, e.g. all elements on the $\mathbf{L}$ diagonal are ones (*Doolittle's Method*), or all elements on $\mathbf{U}$ diagonal are ones (*Crout's Method*). Once triangular factors are known, a solution can be found using forward and backward substitutions (each one requires roughly $n^2/2 = \mathcal{O}(n^2)$ operations). The LU factorisation is mathematically equivalent to Gaussian elimination, therefore if several linear systems differing only on right-hand side vectors have to be solved, it is advisable to perform the LU decomposition and then solve the system for different $\mathbf{b}$ vectors. Again, partial or full pivoting is required to ensure numerical stability. The LU factorisation with partial pivoting is often called LUP decomposition, because it produces an additional permutation matrix $\mathbf{P}$ that satisfies $\mathbf{PA} = \mathbf{LU}$.

If matrix $\mathbf{A}$ is *symmetric* and *positive-definite*, then the *Cholesky decomposition* can be used instead. This method needs only half the number of operations required by the LU decomposition (Press et al., 2007, p. 100), and produces a lower-triangular matrix $\mathbf{L}$ that satisfies $\mathbf{A} = \mathbf{LL}^T$. Another advantage over the LU decomposition, is that the Cholesky method is always numerically stable. If the decomposition fails, it indicates that matrix $\mathbf{A}$ (or, in the presence of round-off errors, another nearby matrix) is not positive-definite. In fact, Cholesky decomposition is an efficient way of determining whether a matrix is positive-definite (Press et al., 2007, p. 101).

2.2.2 Stationary Iterative Methods

A comprehensive overview and analysis of the stationary iterative methods can be found in Barrett et al. (1994) and Briggs et al. (2000). Stationary methods can be

[2] A matrix is *diagonally dominant* if for every row of the matrix, the magnitude of the diagonal entry in a row is larger than the sum of the magnitudes of all the other entries in that row.

[3] Matrix $\mathbf{A}$ is *positive-definite* if for every vector $\mathbf{x} \neq 0$, $\mathbf{x}^T\mathbf{Ax} > 0$.

expressed in a simple form

$$\mathbf{x}^{i+1} = \mathbf{B}\mathbf{x}^i + \mathbf{c}\,, \tag{2.10}$$

where the matrix $\mathbf{B}$ and the vector $\mathbf{c}$ are constant.

A brief overview of stationary iterative methods, based on a summary by Barrett et al. (1994, Ch. 2), is provided below.

Jacobi method is based on solving each variable locally. In each iteration, every variable is solved once with respect to the current value of other variables. All variables are solved *independently*, which makes the Jacobi method easy to implement and enables easy parallelisation, but typically the convergence is slow.

Gauss-Seidel method is similar to the Jacobi method, but new variable values are used as soon as they are available. In consequence, the convergence rate is improved, however non-trivial data dependencies are introduced, and make parallelisation less straightforward.

Weighted Jacobi method is a variant of the Jacobi method (Briggs et al., 2000, p. 9) where each new iterate is a weighted average of the old and new solution vectors. A *factor* ω is introduced to control weights of the average. Briggs et al. (2000, p. 21) prove that $\omega = 2/3$ is the optimal choice when this method is used as a *smoother* in the Multigrid Method (Subsection 2.2.4).

Successive Over-Relaxation (SOR) generalises the Gauss-Seidel method by following the weighted average idea, similar to the Weighted Jacobi method. This method, along with a mathematical framework for iterative methods, was first presented in the PhD dissertation by Young (1950). Choosing the optimal ω value is crucial for the convergence rate of the SOR method, but it is often impractical to compute it in advance. However, some heuristics can be used, e.g. for linear systems coming from discretisation of PDEs, Barrett et al. (1994, p. 10) suggested $\omega = 2 - \mathcal{O}(h)$, where h is the distance between the points in the underlying mesh. If the optimal value for ω is used, then the convergence of the SOR method is an order of magnitude faster than the Gauss-Seidel method.

Symmetric Successive Over-Relaxation (SSOR) is a variant of the SOR for symmetric coefficient matrices $\mathbf{A}$. For optimal values of ω, SSOR usually converges *slower* than the SOR method (Young, 1971, p. 462). In consequence, it is not used as a standalone solution method, but rather as a preconditioner for other iterative schemes for symmetric matrices, e.g. the Conjugate Gradients.

2.2.3 Gradient Methods

Some iterative methods are designed for special cases of linear systems. If the coefficient matrix is *symmetric* and *positive-definite*, then a *quadratic form*

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c\,, \tag{2.11}$$

has a global minimum, which satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$ (Shewchuk, 1994, p. 2). The simplest approach taking advantage of this property is the *Steepest Descent* method. In every iteration, approximate solution $\mathbf{x}^{(i)}$ is modified by a step in the direction with the greatest decrease in the value of f. Once the direction is chosen, the step length is calculated in order to minimise f along that line.

The main shortcoming of the Steepest Descent method, is that it often makes steps in the same direction as in the earlier iterations, which may result in slow convergence. To overcome that limitation, Hestenes and Stiefel (1952) proposed the *Conjugate Gradients* (CG) method, which introduced *conjugate search directions*. In *exact arithmetic* Conjugate Gradients requires at most n iterations to find the exact solution to a linear system with n equations, thus it was first proposed as a direct method. However, this property does not hold on finite-precision computers, therefore the CG method was initially discarded by the mathematical society. It was rediscovered by Reid (1971), who proposed to use it as an iterative method for sparse matrices, rather than a direct method for general matrices. A step-by-step derivation and a detailed analysis of numerical performance of the Steepest Descent and the Conjugate Gradients methods are provided by Shewchuk (1994). For theoretical analysis of the Conjugate Gradients convergence refer to Kaniel (1966), and van der Sluis and van der Vorst (1986).

Many variations of the original Conjugate Gradients method were published. The most popular ones are listed below (after Barrett et al., 1994).

Conjugate Gradients on the Normal Equations (CGNE and CGNR) methods involve application of the CG to both forms of the *normal equations* for system $\mathbf{Ax} = \mathbf{b}$: CGNE solves the system $(\mathbf{AA}^T)\mathbf{y} = \mathbf{b}$, and then computes solution $\mathbf{x} = \mathbf{A}^T\mathbf{y}$; CGNR solves $(\mathbf{A}^T\mathbf{A})\mathbf{x} = \mathbf{A}^T\mathbf{b}$. The CG may be employed in these methods even if coefficient matrix $\mathbf{A}$ is non-symmetric, since matrices $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$ are symmetric and positive-definite.

Bi-conjugate Gradients (BiCG) method generates two sequences of conjugate search direction vectors: for coefficient matrices $\mathbf{A}$ and $\mathbf{A}^T$ (Fletcher, 1976). This method is useful for non-symmetric matrices, however the convergence may be irregular and the method may even break down.

Conjugate Gradients Squared (CGS) method is a further development of BiCG that does not need the multiplications with the transpose of matrix $\mathbf{A}$ (Sonneveld, 1989). Ideally, the convergence rate should be doubled, but in practice it may be even more irregular that for the BiCG.

Bi-conjugate Gradients Stabilised (Bi-CGSTAB) improves the BiCG method by introduction of the search vector sequences re-orthogonalisation (van der Vorst, 1992). In consequence, the convergence is more regular. The Bi-CGSTAB method can be applied to non-symmetric coefficient matrices.

2.2.4 Other Methods

In addition, there are other iterative methods, which are not in the scope of this project, but are listed below for the survey completeness.

Lanczos method (Lanczos, 1952) is an adaptation of power methods to find eigenvalues and eigenvectors proposed by Lanczos (1950), and is based on two mutually orthogonal vector sequences. The most prominent feature of this method is the reduction of the original matrix to tridiagonal form.

Arnoldi algorithm (Arnoldi, 1951) combines Lanczos and Conjugate Gradients approaches since it uses only one, self-orthogonal vector sequence, however it does not use CG-like search directions. The Arnoldi algorithm reduces a matrix to upper Hessenberg form, and can be applied to non-symmetric matrices.

Minimal Residual (MINRES) and **Symmetric LQ** (SYMMLQ) methods (Paige and Saunders, 1975) are Lanczos method variants which can be applied to symmetric, but possibly indefinite linear systems.

Generalised Minimal Residual (GMRES) produces a sequence of orthogonal vectors, and combines them through the least-squares solve and update (Saad and Schultz, 1986). The GMRES method stores the entire vector sequence, thus requires a significant amount of memory. To overcome this problem, implementations with restarting are considered. The GMRES is especially useful when applied to general non-symmetric matrices.

Quasi-Minimal Residual (QMR) method (Freund and Nachtigal, 1991) applies the least-squares solve and update to BiCG residuals, effectively smoothing out the irregular convergence, and avoiding the potential breakdown of the BiCG method. However, the QMR method does not effect a true minimisation, resulting in no significant improvement in performance over the BiCG.

Multigrid Methods is the family of methods operating on multiple meshes of different sizes, instead of just one. These methods are based on two observations:

- simple stationary methods reduce high frequency modes (associated with small eigenvalues) in the error vector fast, however they perform badly on low frequency modes (associated with large eigenvalues),
- low frequency modes on a fine mesh become high frequency modes on a coarser mesh.

In addition to a simple iterative method called a *smoother*, the Multigrid Method requires two operators to map vectors between meshes of different size: *restriction* and *interpolation*. In the classic (*geometric*) Multigrid Method, the underlying mesh is used to construct multilevel hierarchy of meshes; an extension for general linear systems is available — *Algebraic Multigrid* (AMG). In AMG, operators are constructed directly from the coefficient matrix, making it an effective solver for general sparse matrices. For a detailed overview and performance analysis of Multigrid Methods refer to Briggs et al. (2000), and for an introduction to the Algebraic Multigrid method refer to Falgout (2006).

2.2.5 Preconditioning

The spectral properties of the coefficient matrix have strong impact on the convergence rate, and thus the performance, of iterative methods (Barrett et al., 1994, Ch. 3). If matrix $\mathbf{M}$ approximates $\mathbf{A}$, then the linear system

$$\mathbf{M}^{-1}\mathbf{A}x = \mathbf{M}^{-1}\mathbf{b}\,, \tag{2.12}$$

has the same solution, but more favourable spectral properties.

Barrett et al. (1994, Ch. 3) distinguish two types of preconditioners: *implicit*, where matrix $\mathbf{M}$ approximates matrix $\mathbf{A}$ and $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$ is easy to compute, and *explicit*, where matrix $\mathbf{M}$ approximates $\mathbf{A}^{-1}$. The majority of preconditioning methods fall into the former category.

It is important to ensure that the cost of preconditioning is not higher than the benefits from faster convergence. Some preconditioners may require a *construction step*, which is typically proportional to the number of variables. In addition, the construction of some preconditioners has a large sequential component, and the opportunities for parallelisation are limited.

A brief overview of the most popular preconditioning techniques is presented below. For more details refer to Barrett et al. (1994, Ch. 3), Benzi (2002), or Saad (2003, Ch. 10). In addition, Boman and Hendrickson (2003) introduce *support theory*, which provides analytical bounds for condition numbers for preconditioned symmetric and positive-definite linear systems.

Jacobi preconditioner consists of the diagonal of the matrix $\mathbf{A}$. It is easy to implement and works well on parallel computers, however it typically offers smaller improvement in convergence rate than the more sophisticated preconditioners.

Symmetric Successive Over-Relaxation preconditioner can be also easily constructed from the coefficient matrix. SSOR offers good improvement of spectral properties, but only if parameter ω is close to optimal. However, in the general case, calculating optimal ω is prohibitively expensive.

Incomplete LU and Cholesky factorisations are among the most popular preconditioners, however they have significant disadvantages: they can be numerically unstable (Golub and Van Loan, 1996, §10.3), the construction phase is expensive, and they are not suitable for parallelisation.

Polynomial preconditioners directly approximate the inverse of the coefficient matrix, unlike the previous methods. Dubois et al. (1979) investigated the approach based on the *Neumann expansion* of the $\mathbf{A}^{-1}$ matrix, and showed that under certain assumptions, if p elements of the expansion are computed, k steps of the polynomially preconditioned Conjugate Gradients method are equivalent to $k \cdot p$ steps of the original method.

Multigrid preconditioners are based on the Multigrid Methods with a reduced tolerance. Algebraic Multigrid was successfully applied as a preconditioner for the Conjugate Gradients method (Iwamura et al., 2003; Pereira et al., 2006).

2.3 Parallel Computing

The history of parallel computing is almost as long as the history of computers. However, the most rapid development in *shared memory* parallel computing could be observed in the last decade, due to the introduction of widely available multicore CPUs, and the emergence of general-purpose computing on graphics processing units (GPGPU). The theoretical background of parallel computing is presented in Subsection 2.3.1.

An interesting perspective on parallel computing originates from the University of California, Berkeley (Asanovic et al., 2006, 2009). The conclusions presented in these publications highlight challenges facing scientists, engineers, and software developers working with parallel systems.

- The goal should be to make it easy to write software that executes efficiently on highly parallel computing systems.
- The target should be thousands of cores per chip, as these processing units tend to be the most efficient in terms of instruction throughput per watt.
- Auto-tuners should play a larger role than conventional compilers in translating parallel programs.
- To maximise programmer's productivity, future programming models must be more human-centric than the conventional focus on hardware or applications.
- Programming models should be independent of the number of processors.

Every statement presents its own challenges, and a significant scientific and engineering effort is required before they are even partially met. This project is focused on targeting many-core platforms, i.e. effective utilisation of the large number of cores per chip present on modern GPUs. The literature review on the general-purpose computing on graphics processing units (GPGPU) is presented in Subsection 2.3.2.

Typically, CPUs provide several cores capable of performing fairly complex operations. In contrast, GPUs provide hundreds of relatively simple processing units. This makes these platforms well-adjusted to different classes of problems. Modern high-performance computing facilities consist of a large number of multi-core CPUs and many-core GPUs. Existing software needs to be redesigned to take advantage of *hybrid parallelism* in order to fully utilise computational capabilities of such platforms. The literature review on developments and applications of hybrid parallelism is presented in Subsection 2.3.3.

This thesis introduces the Distributed Block Direct Solver (Chapters 7 and 8) — a novel implementation of the Block Relaxation scheme designed to run on hybrid CPU-GPU platforms. DBDS requires a number of parallel building blocks — the publications that helped with the development of the DBDS method are summarised in Subsection 2.3.4.

2.3.1 Theoretical Background

The literature on parallel computing is dominated by applications to particular problems. However, important works have been published on classification of parallel architectures, and theoretical performance of parallel algorithms. The most prominent results are summarised in this subsection.

Flynn's Taxonomy

The early classification of parallel computers was introduced by Flynn (1972). Four types were distinguished, based on the number of instruction and data streams.

Single Instruction, Single Data (SISD) denotes a sequential computer. A single processing unit operates on a single stream of data, one operation at a time. The traditional single-core CPUs are an example of the SISD architecture.

Single Instruction, Multiple Data (SIMD) computers perform the same set of instructions on different sets of data in parallel. Examples of SIMD architectures include vector processors and GPUs.

Multiple Instruction, Single Data (MISD) is an uncommon architecture, typically used for fault tolerance. Heterogeneous systems process the same data, and must agree on the result. The control computer of a space shuttle is an example of the MISD architecture.

Multiple Instruction, Multiple Data (MIMD) denotes a computer comprising a number of independent processing units, concurrently executing different instructions on different sets of data. Distributed systems are generally considered to be MIMD architectures.

The last classification is often further divided into two following categories.

Single Program, Multiple Data (SPMD) is similar to the SIMD model, since the same set of instructions is executed by multiple autonomous processing units. The main difference is that the processing units do not have to perform the same instruction at the same time, i.e. they execute the program independently, rather than in a lock-step.

Multiple Program, Multiple Data (MPMD) consists of multiple autonomous processing units executing at least two different programs. Typically, this model is used to implement the *master-worker* strategy. The processing units executing the *master* code distribute sets of data and gather results from the processing units executing the *worker* program.

This thesis deals with what may be considered the most difficult: MPMD model for the CPU codes combined with SIMD model for the GPU codes.

Theoretical Performance Limits

In the early years of parallel computing, it was believed that if a program executes on one processing unit in time T, then on P processing units it should execute in T/P time. This naive thinking was changed with the publication of *Amdahl's Law* (1967), based on a simple observation that some parts of a program cannot be

parallelised. If the fraction of sequential code is denoted by s, then according to Amdahl's Law, the program can run only up to $1/s$ times faster, regardless of the number of processing units, e.g. if 5% of the program is sequential, then a possible acceleration is bound by a factor of 20.

Amdahl's Law was often criticised for being unnecessarily pessimistic in the assumption that the sequential fraction is constant. In fact, in many real-life applications, the sequential fraction is decreasing with increasing problem size. Gustafson pointed out that usually the goal is not to solve a fixed size problem in the shortest possible time, but rather to solve the largest possible problem in a fixed time. *Gustafson's Law* (1988) was proposed as an alternative bound for parallel performance. It states that if the time spent executing the sequential code is fixed, then the problem of an arbitrary size can be solved in a fixed period of time, provided that sufficient number of processing units is available.

Another tool used to assess parallel algorithm performance is the *Karp-Flatt Metric* (1990), which allows us to estimate the fraction of sequential code. This metric allows us to take parallelisation overheads into account, which are omitted by the two above-mentioned laws. When applied to a fixed-size problem, the Karp-Flatt Metric determines the source of degradation in parallel efficiency observed when the number of processing units is increased — it may be due to limited parallelisation opportunities, or may be connected with algorithmic or architectural overheads.

Parallel Computing Models

A parallel computing model is a conceptual view of the memory organisation and types of operations available to a program. Each model can be implemented on any hardware architecture, if adequate support is provided by the operating system. However, the performance of the implementation greatly relies on suitability of the underlying hardware, i.e. models are devised to perform well on particular hardware platforms. A brief overview of the two most popular models is presented below.

The *message-passing* model defines that the communication between processing units is done via messages exchanged over the underlying network. Every processing unit has its own local memory. This model finds application mainly in heterogeneous computer clusters, typically connected with relatively slow, high-latency networks.

The most popular implementation of the message-passing model is *Message Passing Interface* (MPI, Gropp et al., 1999) — a language-independent communications protocol, specifying types of messages available to programs. It supports *point-to-point* (e.g. a master node sending data to a worker node) and *collective* (e.g. finding the maximum value among the results returned by the processing units) operations.

An alternative set of tools and libraries for general-purpose, heterogeneous, concurrent computing, based on the message-passing model is the *Parallel Virtual Machine* (PVM) system (`http://www.csm.ornl.gov/pvm/`). PVM documentation, users' guide, and tutorials are provided by Geist (1994).

The *data parallelism* model aims to take advantage of the data dependencies. The program designed in that model looks similar to a sequential code, but blocks which are likely to enable parallel computing (e.g. `for` loops) can be annotated appropriately. When such a block of code is reached, the data is partitioned into independent sets and processed in parallel. This model often assumes the presence of *shared memory*, and it is typically used on the modern multi-core CPUs, where multiple concurrent threads are used to handle parallel blocks. The most popular API for the data parallelism model is *Open Multi-Processing* (OpenMP, `http://openmp.org/`). Chandra et al. (2001) provide more information on OpenMP programming.

The most recent direction in parallel computing is *general-purpose computing on graphics processing units*. The GPGPU is a technique of employing GPUs to perform computation traditionally handled by the CPUs. The literature review on the GPGPU developments and applications is provided in Subsection 2.3.2.

Finally, *hybrid parallel programming* combines the above-mentioned models. The most common approach is to employ message-passing for coarse-grain parallelism, in combination with a shared memory model for fine-grain parallelism. The literature review on hybrid parallelism is provided in Subsection 2.3.3.

2.3.2 General Purpose Programming on the GPU

Initially, GPUs were used to handle computation required to display computer graphics. The insatiable market demand for real-time, high-definition 3D graphics in computer games was one of the major factors driving the rapid development in

GPU hardware. In early 2000s the speed of graphics cards exceeded the performance of CPUs and the research on application of GPUs in general-purpose computing began. However, computer games legacy could be seen in graphics hardware used for GPGPU for a long time, e.g. a significantly lower performance, or even no support, for double precision arithmetic (not necessary for displaying computer graphics).

Graphics processors provide the first widely available many-core platform. The first successful attempts to utilise GPUs as co-processors were reported in early 2000s, e.g. Conjugate Gradients and Multigrid solvers by Bolz et al. (2003) and Goodnight et al. (2003). However, it was not until NVIDIA (a graphics hardware vendor) released the CUDA platform (Nickolls et al., 2008) in February 2007 and Tesla hardware (Lindholm et al., 2008), when popularity of the GPGPU exploded. For more details on pre-CUDA GPGPU works refer to a survey by Owens et al. (2007). For the details on the early days of the GPU computing with CUDA refer to a survey by Garland et al. (2008), and two excellent books by Sanders and Kandrot (2010) and Kirk et al. (2010).

The main disadvantage of the CUDA platform is that it works only with NVIDIA's hardware. Therefore, soon after the initial release of CUDA, work began on *Open Computing Language* (OpenCL, `http://www.khronos.org/opencl/`), which was released in December 2008. The main aim was to create a framework for writing programs that execute across heterogeneous platforms comprising CPUs and GPUs.

The developments in the GPGPU created the possibility of porting classic algorithms to powerful, modern graphics cards, which raised significant interest among researchers. This resulted in numerous publications reporting execution times an order of magnitude faster than the CPU implementations. These applications include the Conjugate Gradients solver (Stock and Koch, 2010; Wozniak et al., 2010), tridiagonal solvers (Zhang et al., 2010; Göddeke and Strzodka, 2011), solving the Euler equations to simulate a hypersonic vehicle configuration (Elsen et al., 2008), a Multigrid solver used in the power grid analysis (Feng and Li, 2008), and our own work — the Tabu Search meta-heuristic for various $\mathcal{NP}$-hard optimisation problems (Czapiński and Barnes, 2011; Czapiński, 2013).

After the initial enthusiasm and reports of GPU implementations tens, or even hundreds, times faster than their CPU counterparts, came the reflection that these numbers cannot be right when compared to the physical capabilities of these platforms.

Indeed, the most popular metric — GPU-CPU speed-up ratio is very sensitive and becomes biased if both CPU and GPU codes are not fully optimised. As this was often the case, Lee et al. (2010) prepared highly optimised codes for several applications, and performed rigorous performance comparison between Intel Core i7-960 CPU and NVIDIA GeForce GTX 280 GPU. The authors report that on average the GPU was only 2.5 times faster than the CPU, however the speed-ups varied greatly for different applications. The important conclusion was that the speed-up ratio is not a reliable metric, and performance of GPU implementations should be measured with instruction or memory throughput, depending on whether the problem is compute- or memory-bound. Then reliable assessment can be made based on what percentage of the theoretical peak performance of the device was obtained.

In an IBM technical report, Bordawekar et al. (2010) go even further in their scepticism about the GPUs' performance. The authors compared the NVIDIA GeForce GTX 285 with the Intel Xeon (Nehalem) and the IBM Power7 on an image cross-correlation algorithm, and claimed that IBM Power7 was indeed the fastest platform. However, the presented results could be biased: reported run times were around one second, suggesting relatively small problems, for which the GPU might have only used a fraction of its computing resources.

Developing CUDA applications is far from the ideal sketched by Asanovic et al. (2006). Che et al. (2008) presented a discussion on shortcomings of CUDA, especially low-level memory management. The authors conclude that there is a need for a high-level API for more complex data structures and parallel programming primitives. Moreover, Ryoo et al. (2008) attempted to create some general parameter optimisation strategies for CUDA. The results were not promising, as only simple rules provided any performance improvement. No general rule could provide a consistent improvement in performance on considered benchmarks. However, the authors managed to obtain good performance with hand-crafted codes.

The above-mentioned shortcomings were recognised and numerous attempts to alleviate them were proposed. Among them are the highly optimised implementations of *parallel scan* (Harris et al., 2007) and *parallel reduction* (Harris, 2008). These operations are trivial to program on sequential platforms, but efficient implementation on parallel architectures is challenging. In addition, they are the building

blocks in many applications, e.g. parallel scan can be used to implement sorting or shallow-water fluid simulation (Sengupta et al., 2007).

Additional parallel building blocks are available in the form of software libraries, e.g. Fast Fourier Transform (CUFFT, NVIDIA, 2012b), and basic routines for dense (CUBLAS, NVIDIA, 2012a) and sparse (CUSPARSE, NVIDIA, 2014) algebra. Moreover, the MAGMA library (`http://icl.cs.utk.edu/magma/`) provides highly optimised CUDA implementations of Linear Algebra PACKage (LAPACK; Anderson et al., 1999) routines, e.g. implementation of dense matrix-matrix multiplication (Nath et al., 2010), which delivers outstanding performance of up to 645 GFLOPS (single precision) and 300 GFLOPS (double precision) on NVIDIA Tesla C2050 (roughly 60% of the theoretical peak performance). Furthermore, the authors propose the GPU implementation of the LU factorisation, which achieves a higher fraction of the GPU peak performance than the state-of-the-art implementations on high-end x86-based multi-core systems. More GPU libraries can be found at `http://developer.nvidia.com/technologies/Libraries/`.

For researchers working with PDEs, sparse algebra routines are at the core of interest. Unlike dense algebra, which is typically compute-bound, sparse algebra operations are more often memory-bound. Data transfer is still the main bottleneck for modern GPUs, which makes it more difficult to develop efficient GPGPU implementations. Despite this limitation, there are a number of publications presenting different techniques to speed up calculations in sparse linear solvers (Buatois et al., 2009; Stock and Koch, 2010; Wozniak et al., 2010).

Many researchers identified the sparse matrix-vector multiplication (SpMV) as the most time-consuming part of sparse linear solvers, e.g. the Conjugate Gradients method, and focused their efforts on optimising this routine (Kourtis et al., 2008; Baskaran and Bordawekar, 2009; Williams et al., 2009; Guo and Wang, 2010; Li et al., 2013). Bell and Garland (2008) ran extensive experiments on several implementations and different sparse matrix storage formats. Their implementations were able to utilise up to 63.5% of the peak memory bandwidth, however the instruction throughput was low — only 36 GFLOPS (single precision) and 16 GFLOPS (double precision) on NVIDIA GeForce GTX 280.

Since the SpMV operation is memory-bound, Willcock and Lumsdaine (2006) and Kourtis et al. (2008) suggested that compression of the sparse matrix could speed up

computation. Moreover, Pichel et al. (2008) investigated how to utilise the SIMD architecture better by reordering data, and thus improving the SpMV performance. Despite an additional computation cost, both methods resulted in speed-ups of calculations up to 30% on Intel Core2Duo and Xeon architectures, however no results have been reported for GPUs. Nevertheless, these techniques may be applicable and able to improve the performance on the graphics hardware.

Finally, Stratton et al. (2008) proposed MCUDA — a library that allows running CUDA applications on shared memory, multi-core CPUs. The experiments confirmed good performance, and demonstrated that CUDA can be an effective data-parallel programming model for more than just GPU hardware.

2.3.3 Hybrid Parallelism

In the presence of clusters with many multi-core processors, and high-performance supercomputers comprising a large number of CPUs and GPUs, only the hybrid parallel computing model allows us to take full advantage of the available computational power. However, a combination of the different models is difficult in terms of software complexity, and keeping the parallelisation overheads at a low level.

After the early papers reporting good performance results on a single GPU, the natural next step was to employ multiple GPUs, adding the second level of parallelism: first, the problem has to be divided into subproblems that can be solved simultaneously, and then each subproblem has to be processed using many cores of the GPU. These hybrid parallel architectures typically run one CPU thread per GPU, and use MPI or OpenMP to manage the execution. The GPUs may reside on the same motherboard, or in separate systems. The main benefit of the former approach is fast communication via shared memory, but only a few (typically up to four) GPUs can connect to the same motherboard, effectively limiting the scalability. In the latter case, there is no such limitation, but this approach may suffer from higher latency and possibly smaller data transfer rates, e.g. an order of magnitude lower bandwidth in the case of Gigabit Ethernet. In consequence, efficient communication is crucial for the performance of any hybrid algorithm.

In the literature, different types of hybrid models have been considered: MPI-OpenMP (Quinn, 2004), MPI-CUDA, and OpenMP-CUDA (Yang et al., 2011). In

an attempt to make hybrid parallel implementations easier, Lawlor (2009) proposed cudaMPI — the library supporting communication in the MPI-CUDA model. However, cudaMPI only wraps the MPI and CUDA APIs, and does not offer any gain in performance or the level of flexibility available when these APIs are used directly.

Most hybrid parallel algorithms reported in the literature were developed for and tested on multiple GPUs connected to the same motherboard. These include sparse matrix-vector multiplication used in the Conjugate Gradients solver (Cevahir et al., 2009), LU factorisation (Tomov et al., 2010a), matrix-matrix multiplication, MD5 hashing and sorting (Yang et al., 2011), and Roe's method for two-layer shallow water systems (de la Asunción et al., 2012). The results presented by Cevahir et al. (2009) show that even when shared memory is used, communication may become a bottleneck — the proposed solver obtains up to 11.6 GFLOPS on a single GPU, and only 24.6 GFLOPS on four GPUs, which is far from the desired linear speed-up.

Overlapping communication and computation has been widely used to improve the performance of parallel algorithms following the message-passing model (Brightwell et al., 2005; Hoefler et al., 2007; Shet et al., 2008; Thakur and Gropp, 2009). Since the CUDA API supports non-blocking (asynchronous) communication (NVIDIA, 2011), it is possible to apply this technique in the hybrid MPI-CUDA algorithms. One of relatively few attempts so far, was reported by White and Dongarra (2011), where a hybrid implementation of explicit time integration of linear advection was considered. Micikevicius (2009) described single- and multi-GPU parallelisation of the stencil operator for 3D finite difference computation. Both publications use the Infiniband for connectivity, and report almost linear speed-ups with respect to the number of GPUs. There are no reports of efficient multi-GPU parallel algorithms on architectures with slower connectivity, e.g. Fast and Gigabit Ethernet. The possibility of creating such algorithms is explored further in Chapters 3 and 4.

Among the attempts to utilise hybrid parallelism, is the software package FEAST, supporting FEM calculations for 2D Poisson's Equation on multiple GPUs (Göddeke et al., 2007a, 2008). The experiments showed good performance-cost ratio, and scalability, even on heterogeneous platforms. Further studies showed that speeding up the Navier-Stokes solver proved to be more difficult — on a small GPU cluster, the speed-ups barely exceeded a factor of two (Göddeke et al., 2009).

Another attempt for hybrid parallel computations for incompressible flow (Navier-Stokes Equations) was made by Jacobsen et al. (2010). The proposed solver took advantage of overlapping computation and communication, and delivered sustainable 2.4 TFLOPS on 64 nodes with 128 GPUs (a total of 30,720 computing elements). A computational fluid dynamics simulation with 8 billion grid cells was accelerated by a factor of 130, in comparison to a pthreads version on two quad-core Intel Xeon 2.33 GHz CPUs. Furthermore, the proposed implementation provided good weak scalability — roughly 58% parallel efficiency was observed on 128 GPUs.

Pennycook et al. (2011) investigated the performance of hybrid parallel codes, by porting the LU benchmark from the NAS Parallel Benchmark (Bailey et al., 1991) to the MPI-CUDA model. The experiments were performed on a range of GPUs: from low-end to HPC-grade hardware (Tesla C2050). The main conclusion from this study is that achieving performance at scale on multi-GPU clusters is challenging due to increasing communication cost. Furthermore, developers will face a dilemma between low-level code optimisations for local clusters, and focusing on high-level distributed scalability for large quantities of independently operating cores.

In an attempt to make the development of hybrid parallel programs easier, the Hybrid Multi-core Parallel Programming Environment (HMPP) was proposed (Dolbeau et al., 2007; Bodin and Bihan, 2009). The HMPP allows for seamless transition of legacy code using OpenMP-style pragmas to utilise hybrid resources. However, it is difficult to assess the performance of codes generated by HMPP — the authors reported 2–10 times speed-ups against the CPU implementation, but the conditions of experiments (e.g. the CPU model) were not clearly specified.

Alonso et al. (2011) proposed a linear solver based on Neville elimination, and compared OpenMP-CUDA and MPI-CUDA models. The experiments showed that MPI is typically faster for inter-node communication. However, significant differences between different MPI implementations were observed: HP-MPI proved to be much faster than MPICH, which was faster than OpenMPI.

2.3.4 Towards the DBDS Solver

Algorithms following the Block Relaxation scheme, also known as the *group iterative* methods (Young, 1971, Ch. 14), were used to solve sparse linear systems since the

1970s. This approach is a generalisation of the classic stationary iterative schemes (Jacobi, Gauss-Seidel, SOR; cf. Subsection 2.2.2), which may be considered "point" relaxation methods. In the Block Relaxation scheme, a whole set of solution vector components are updated at each time, rather than just one component. For detailed description of Block Relaxation refer to the above-mentioned textbook by Young (1971) and to works by (Varga, 1999, §6.4) and (Saad, 2003, §4.1.1).

Recently, the GPU implementations of the algorithms following the Block Relaxation scheme have been considered (Anzt et al., 2012; Rodriguez et al., 2012). However, these works focused on stencil operators, i.e. structured sparse matrices only.

Among the recently published algorithms is the *Distributive Conjugate Gradient* (DCG) solver (Becker, 2006; Becker and Thompson, 2006), based on the classic Conjugate Gradients method. The DCG employs the Additive Schwarz domain decomposition, which allows for almost independent parallel solution of the subproblems, effectively decreasing the communication cost. However, the most important property (mutual conjugacy of the search directions) of the CG method is not maintained. Published empirical results showed good DCG performance for several equations, but no analysis of the theoretical properties was provided. Nevertheless, the design of the DCG method is suitable for hybrid parallelism, and will be considered in this project.

The DBDS method following the Block Relaxation scheme (Chapter 7) provides a flexible framework for linear systems solution. By exchanging DBDS components and using extensions, the solver can be easily optimised for a particular problem. Literature supporting the efficient implementation is discussed below.

Matrix Factorisations

Volkov and Demmel (2008) proposed one of the first GPU implementations of the LU, Cholesky, and QR factorisations. This work was extended by Tomov et al. (2010b) who proposed hybrid parallel implementations of all three decompositions. Nath et al. (2010) proposed a very fast implementation of the LU factorisation on the GPU, notably achieving a significant fraction of the theoretical peak performance.

However, all the above-mentioned implementations were designed for dense matrices. The first attempt to speed-up direct sparse factorisation was made by Christen et al.

(2007). The authors followed a supernodal approach, and identified dense matrix-matrix multiplication (GEMM) as a dominant operation. Their approach was to offload GEMM operations to the GPU for sufficiently large matrices. Indeed, this resulted in performance improvement by a factor of 1.5–3.5 depending on the matrix sparsity pattern (experiments were run on a GeForce 8800 GPU and an Intel Pentium 4 3.4 GHz CPU). The second idea was to perform a sparse factorisation entirely on the GPU, however this proved impractical due to the relatively small memory.

Other published direct sparse solvers with GPU acceleration followed a multifrontal approach to factorisation (Duff and Reid, 1983). In this case, multiple decompositions have to be performed on smaller *frontal* matrices. These are dense and efficient GPU implementations mentioned above could be used. Krawezik and Poole (2010) performed experiments on the BM-7 wing model from the ANSYS benchmarks and reported speed-ups up to a factor of 2.9 in double precision and up to a factor of 4 in mixed precision (see below). The experiments were performed on a Tesla C1060 GPU and an Intel Xeon 5335 2.0 GHz CPU.

Yu et al. (2011) performed experiments on a wider range of non-symmetric linear systems and focused mostly on optimising CPU-GPU memory transfers. On a Tesla C2070 GPU, this allowed the authors to speed-up sparse factorisation up to a factor of 3.45 in comparison to the Intel MKL library running on an Intel Xeon E5620 CPU. However, the performance was highly irregular — the GPU accelerated implementation was faster for only 9 out of 18 realistic problems. Noticeably better performance was observed on five artificial linear systems (speed-ups of a factor 3.9–4.5), which have considerably more non-zero elements.

George et al. (2011) introduced an auto-tuning capability to a multifrontal direct factorisation for SPD matrices. This allows the authors to perform the sparse factorisation up to 7 times faster on two T10 GPUs (part of a Tesla S1070 computing system) in comparison to a state-of-the-art WSMP implementation (Gupta and Avron, 2000) running on a single core of an Intel Xeon 5160 CPU.

Three important points should be noted. First, due to limited parallelisation opportunities in the sparse matrix factorisation it is difficult to fully utilise the computational capabilities of the graphics cards. Indeed, the speed-ups reported in the literature are far from the relative theoretical peak performance of CPUs and GPUs

used in experiments. Secondly, the performance of a GPU accelerated factorisation highly depends on the sparsity pattern of a matrix, rather than on its size or the number of non-zero elements. In consequence, it is difficult to predict the execution time on the GPU. Finally, none of the above-mentioned publications considered running the solution step of a direct method (solving two triangular matrices) on the GPU. Instead, this operation was performed on the CPU.

Mixed-precision Iterative Improvement

The concept of the *mixed-precision iterative improvement* (refinement) was first studied by Wilkinson (1965) and Martin et al. (1966). The results presented in the latter paper were then extended by Moler (1967). Iterative refinement aims at improving the accuracy of numerical solutions to linear systems, and allows us to perform most of the computation in low-precision arithmetic and maintain high-precision of the final result.

Langou et al. (2006) enhanced classic direct and iterative solvers with iterative refinement. On an IBM Cell processor, the authors observed up to 10 times speed-up over double precision solvers. Possible application of the mixed-precision iterative improvement method in various sparse iterative techniques on GPUs and Field Programmable Gate Arrays (FPGA) was also investigated by Buttari et al. (2008).

A trade-off for using a faster, low-precision arithmetic is that for ill-conditioned linear systems the mixed-precision approach may become numerically unstable and break down. Langou et al. (2006) indicate that the method works for matrices with a condition number not exceeding 10^8.

On many GPUs, the instruction throughput in double precision is significantly slower (up to 10 times) than in single precision arithmetic. This created a natural application for an iterative refinement technique and was investigated by Göddeke et al. (2007b). The existing mixed-precision methods were compared to solvers for systems arising from a FEM discretisation of PDEs. The mixed-precision iterative refinement allowed the authors to offload up to 99% of the computational effort to a low-precision co-processor (GPU), to achieve speed-ups between four and five times, and to reduce the space requirements significantly.

Moreover, the mixed-precision iterative improvement has been successfully applied in GPU implementations of the Multigrid solver (Göddeke et al., 2008), sparse matrix-vector multiplication (Cevahir et al., 2009), and the Cyclic Reduction tridiagonal solver (Göddeke and Strzodka, 2011).

Tridiagonal Solvers

On a sequential machine, the algorithm by Thomas (1949) finds a solution to a tridiagonal system in $\mathcal{O}(n)$ time. One of the early works on parallel tridiagonal solvers (Stone, 1973) shows that it is possible to achieve logarithmic complexity.

More recently, several GPU implementations have been proposed. Zhang et al. (2010) implemented and compared a few methods: Cyclic Reduction, Parallel Cyclic Reduction, Recursive Doubling, and their hybrids. The experiments confirmed that the fastest proposed implementation on a GeForce GTX 280 GPU was up to 28 times faster than the sequential LAPACK solver, and 12 times faster than the multi-threaded implementation on an Intel Core 2 Q9300 2.5 GHz quad-core CPU (the peak theoretical performance of the GPU was roughly 30 times greater than that of the CPU). Göddeke and Strzodka (2011) proposed an improved GPU implementation of the Cyclic Reduction tridiagonal solver, and provide a detailed analysis of memory access patterns and their impact on the overall performance.

Part I

Advanced Parallel Architectures

The chapters in this part present three case studies concerning capabilities of modern graphics cards and their applicability to solve computationally demanding problems. GPUs provide hundreds of processing cores, however to work efficiently the data must be transferred at high rates. The inability to meet the data throughput requirements often becomes the main performance limiting factor. Therefore, the following three chapters focus on techniques to maximise the data transfer rates.

Chapter 3 explores the possibility of using hybrid CPU-GPU parallelism in a simple PDE solver, namely the Jacobi method for 2D Laplace's Equation. The main focus is on techniques enabling efficient multi-GPU processing in the presence of slow inter-GPU interconnect and heterogeneous graphics cards.

Unlike the Jacobi solver, the Successive Over-Relaxation method considered in Chapter 4 involves non-trivial data dependencies. In consequence, new techniques addressing this issue are proposed and analysed. Furthermore, the SOR method has an order of magnitude better convergence rate than the Jacobi method and can be used as a stand-alone solver. Therefore, it was useful to extend the multi-GPU SOR implementation to solve a more general 2D Poisson's Equation.

The final case study (Chapter 5) presents how to implement more complex data structures on the GPU. The Tabu Search is among the most prominent optimisation meta-heuristics, but requires maintaining a data structure called the *Tabu List*. This task is relatively easy on CPU-based architectures, however it is non-trivial on a graphics card. Thanks to the novel GPU implementation of the Tabu List and various memory throughput optimisations, the proposed Parallel Multistart Tabu Search applied to a well-known $\mathcal{NP}$-hard Quadratic Assignment Problem is many times faster than the multi-threaded CPU implementations and provides solutions with quality matching that of recent state-of-the-art meta-heuristics.

Chapter 3

Case Study: Hybrid Jacobi Solver for 2D Laplace's Equation

This chapter presents a number of techniques that help to reduce communication overhead on hybrid CPU-GPU platforms, especially in environments with relatively slow interconnect. The optimisations include careful data layout and usage of GPU memory spaces, and overlapping computation and communication. The proposed techniques were validated in the Hybrid Jacobi Solver for 2D Laplace's Equation. This model problem was chosen because of its theoretical importance, and low computation-to-communication ratio, which makes efficient parallelisation difficult. Furthermore, an accurate dynamic load balancing technique for heterogeneous multi-GPU environments is proposed.

The results presented in this chapter have been published in the International Journal of Parallel Programming (Czapiński et al., 2013).

3.1 Introduction

When working with multi-GPU platforms, two levels of parallelism must be considered. First, the problem has to be divided into subproblems that can be solved simultaneously, and then each subproblem has to be solved using many cores of the graphics card. The GPUs may reside on the same motherboard, communicating via shared memory (POSIX threads, OpenMP), but only a few devices (typically up to

four) can connect to the same motherboard, effectively limiting the scalability. In the case of distributed memory (MPI), there is no such limitation, but this approach suffers from slower communication over the network (e.g. Ethernet or Infiniband).

In an attempt to make hybrid parallel implementations easier, Lawlor (2009) proposed cudaMPI — the library supporting communication in the MPI-CUDA model. However, cudaMPI only wraps the MPI and CUDA APIs, and does not offer any gain in performance or the level of flexibility available when these are used directly.

Most hybrid parallel algorithms reported in the literature were developed for and tested on multiple GPUs connected to the same motherboard. These include sparse matrix-vector multiplication used in the Conjugate Gradient solver (Cevahir et al., 2009), LU factorisation (Tomov et al., 2010a), and matrix multiplication, MD5 hashing and sorting (Yang et al., 2011). The results presented by Cevahir et al. show that, even when shared memory is used, communication may become a bottleneck — the proposed solver obtains up to 11.6 GFLOPS on one, and up to 24.6 GFLOPS on four GPUs, which is far from the desired linear speed-up.

Overlapping communication and computation has been widely used to improve performance of parallel algorithms following the message-passing model (Brightwell et al., 2005; Hoefler et al., 2007; Shet et al., 2008; Thakur and Gropp, 2009). Since the CUDA API supports non-blocking (asynchronous) communication (NVIDIA, 2011), it is possible to apply this technique in hybrid CUDA-MPI algorithms. One attempt was recently reported by White and Dongarra (2011), where a hybrid implementation of explicit time integration of linear advection was considered. Micikevicius (2009) described single- and multi-GPU parallelisation of the stencil operator for 3D finite difference computation. Both publications report almost linear speed-ups with respect to the number of GPUs, using Infiniband for connectivity. To the best of our knowledge, there are no reports of efficient multi-GPU parallel algorithms on architectures with slower interconnect (e.g. Fast and Gigabit Ethernet).

3.1.1 Model Problem

The 2D Laplace's Equation is defined as follows:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0\,. \tag{3.1}$$

In this study, Equation (3.1) was discretised with the FDM on the unit square $[0,1] \times [0,1]$ using a uniform $n \times n$ grid with the following boundary conditions:

$$\phi(x,y) = \begin{cases} \sin^2(\pi y) & \text{if } x = 0, \\ 0 & \text{if } x = 1,\ y = 0, \text{ or } y = 1. \end{cases} \tag{3.2}$$

The solution to this problem is presented in Figure C.1 B in Appendix C.

A solver following the Jacobi method approximates function ϕ with a grid v. In iteration $(r+1)$ the grid values are updated according to the formula

$$v_{i,j}^{(r+1)} = \frac{1}{4}\left(v_{i-1,j}^{(r)} + v_{i+1,j}^{(r)} + v_{i,j-1}^{(r)} + v_{i,j+1}^{(r)}\right), \quad \text{where } 1 \leq i,j \leq n, \tag{3.3}$$

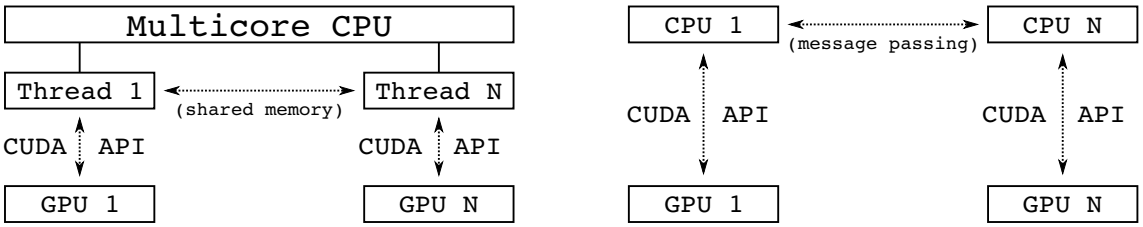
which corresponds to the classic five-point stencil operator.

To converge the Jacobi solver for Laplace's Equation typically requires a number of iterations proportional to the grid size, which is not optimal (Demmel, 1997). In consequence, it does not make a good stand-alone solver, however it is useful in the *smoothing* phase of the Multigrid solver (Briggs et al., 2000).

3.1.2 Hybrid Parallel Architectures

There are several ways of connecting multiple GPUs together, differing on performance and scalability. The first model involves multiple GPUs sharing one motherboard (Figure 3.1 A). Typically one CPU thread is run per GPU, and communication is conducted via shared memory. The main advantage of this model is its simplicity, and fast communication between the GPUs. However, this model is not scalable and typically involves systems with only up to four graphics cards.

Modern high-performance computing installations usually consist of hundreds of CPUs and GPUs, connected with a fast network, e.g. Infiniband. In this model, GPUs are connected to separate motherboards (Figure 3.1 B) and operate on local memory, therefore a message-passing model for communication is used instead. The scalability of such systems is considerably better, but at the cost of more complex and slower communication.



(A) One motherboard (B) Separate motherboards

Figure 3.1: Comparison of typical hybrid multi-GPU architectures.

3.1.3 CUDA Kernels for Jacobi Solver for Laplace's Equation

Five GPU kernels utilising different memory spaces were implemented using the CUDA platform. For the explanation of technical terms and the comparison of different generations of graphics cards (Compute Capability) refer to Appendix A and to the CUDA C Programming Guide (NVIDIA, 2011).

Since our model problem is memory-bound, the optimisation effort was focused on maximising the memory throughput. To achieve that, memory for the grid was allocated with the `cudaMallocPitch` routine. This ensures that each grid row is properly aligned in order to maximise the level of global memory coalescing.

To maximise the speed of CPU-GPU transfers, storage areas in the main memory were *page-locked*, i.e. allocated with the `cudaMallocHost` routine. This is also required to perform memory transfers concurrently with computation.

Each iteration on an $n \times n$ grid requires $3n^2$ additions and n^2 multiplications ($\mathcal{O}(n^2)$ time). Since there are no data dependencies, grid updates can be done in a *constant* number of steps if n^2 processing elements (i.e. CUDA cores) are available.

Global Memory Kernel

In this simplest implementation, each thread reads four adjacent values directly from *global memory*, then computes the new value according to Equation (3.3), and stores it back to global memory.

Accessing a single value in global memory can be expensive, while the latency is typically 300–600 GPU cycles. To compensate for this limitation, global memory

is accessed in larger transactions that allow for *coalescing* multiple reads or writes into fewer operations. This is possible only if certain access pattern requirements are met — these depend on the Compute Capability of the GPU.

On devices with Compute Capability 1.1, to achieve coalesced global memory access, threads in a *half-warp* (16 threads) must access consecutive addresses that are properly aligned. While the first requirement is easily met, it is impossible to meet the alignment requirement for all the reads. In consequence, only limited coalescing can be achieved on devices with Compute Capability 1.1.

On devices with Compute Capabilities 1.2 and 1.3, the coalescing requirements are significantly loosened: it is sufficient that threads in a half-warp access addresses that are within a 128-bytes block, but the alignment requirement holds. The coalescing mechanism on devices with Compute Capability 2.0 is similar, but the introduction of global memory cache can significantly reduce the number of global memory reads.

Shared Memory Kernel

In the optimal implementation, every grid element should be read only once. In the Global Memory Kernel every element is explicitly accessed four times, but the requested value could be read from the cache instead. In the Shared Memory Kernel, caching is implemented manually using *shared memory*. This kernel is mostly aimed at devices with Compute Capability 1.x, as they do not provide any implicit caching for global memory.

In the Shared Memory Kernel, threads are organised into 2D blocks that are mapping corresponding rectangular grid chunks. Before any computation is performed, all threads in the same block copy the corresponding values from global, to a much faster shared memory. Then all threads proceed with calculations. No bank conflicts occur since all threads access consecutive values in shared memory.

The main drawback of the Shared Memory Kernel is that values on block borders have to be loaded as well, causing some threads to access global memory twice. However, only values on block borders amount to excess global memory reads — other values are read only once.

Texture and Surface Memory Kernels

Another way of providing global memory caching, is to use Texture Units. Use of *texture memory* provides a performance boost for data reads which are spatially local — that is the case of the Jacobi solver. The main advantage of the Texture Memory Kernel is that reads do not have to meet coalescing requirements. On the other hand, the texture size is limited to 128 MB, and since texture memory is read-only, the computed values have to be written directly to global memory.

Devices with Compute Capability 2.0 introduce *surface memory*, which also provides caching, but at the same time allows for write operations. The proposed Surface Memory Kernel is similar to the Texture Memory Kernel, but writes computed values directly to surface memory. To the best of our knowledge, this is the first reported application of surface memory to general purpose computation on the GPU.

Finally, a hybrid kernel that combines reads from texture memory and writes to surface memory was implemented. Surface memory can be only bound to the `cudaArray` structure, therefore 2D textures had to be used instead of 1D. According to (Sanders and Kandrot, 2010, Ch. 7), there should not be any significant difference in performance between 1D and 2D textures.

3.1.4 The Blocking Hybrid Solver

In this study, the domain was decomposed into horizontal stripes, one for each GPU. The communication was maintained through the overlapping (*halo*) nodes. Each iteration consisted of updating the values in the local stripe (with kernels presented above), and then exchanging the halo nodes. In the simplest implementation, computation and memory transfers are serialised (Algorithm 3.1). In this case, the GPUs are idle during memory transfers (lines 4–6).

3.1.5 The Non-blocking Hybrid Solver

Most CUDA devices are equipped with at least one asynchronous copy engine, enabling support for the overlapping of the kernel execution and CPU-GPU memory

Algorithm 3.1 The pseudo-code for each node of the blocking Hybrid Jacobi Solver

```
1: Transfer local grid from CPU to GPU memory
2: for r = 0, 1, 2, ... do
3:     Compute grid v^(r+1) using grid v^(r) and the five-point stencil operator
4:     Transfer halo nodes from GPU to CPU memory
5:     Exchange halo nodes with adjacent CPUs
6:     Transfer updated halo nodes from CPU to GPU memory
7: end for
8: Transfer local grid from GPU to CPU memory
```

transfers. To achieve overlapping, multiple CUDA streams, page-locked host memory, and asynchronous memory copy API calls have to be used (NVIDIA, 2011). In the *non-blocking communication* model, border nodes, which are halo nodes in adjacent subdomains, are updated first. Then the border nodes are exchanged, while the rest of the grid is being updated (Figure 3.2). Typical timelines in both communication models are compared in Figure 3.3.

While the same number of floating-point operations is performed in both communication models, the non-blocking model imposes an additional kernel dispatch overhead. Furthermore, Lawlor (2009) reports that the setup time for the asynchronous memory transfer on devices with Compute Capability 1.x is roughly five times longer than in the synchronous mode. That difference disappears in more recent GPUs.

3.2 Numerical Experiments

Technical details of three GPU models (Tesla C1060, Tesla C2050, GTX 480) used in the experiments are summarised in Table B.2 in Appendix B. All installations were using 285.05.09 driver and CUDA Toolkit 4.0. Unless specified otherwise, solvers were run for 1,000 iterations, all experiments were repeated ten times, and the ratio of the standard deviation to the average run time was confirmed to be less than 5%.

In devices with Compute Capability 1.x, global memory coalescing is handled for each half-warp, i.e. 16 threads (NVIDIA, 2011), therefore the usual practice is to use 16×16 thread blocks (Micikevicius, 2009). Since devices with Compute Capability 2.x handle global memory coalescing for the whole warp, all CUDA kernels were

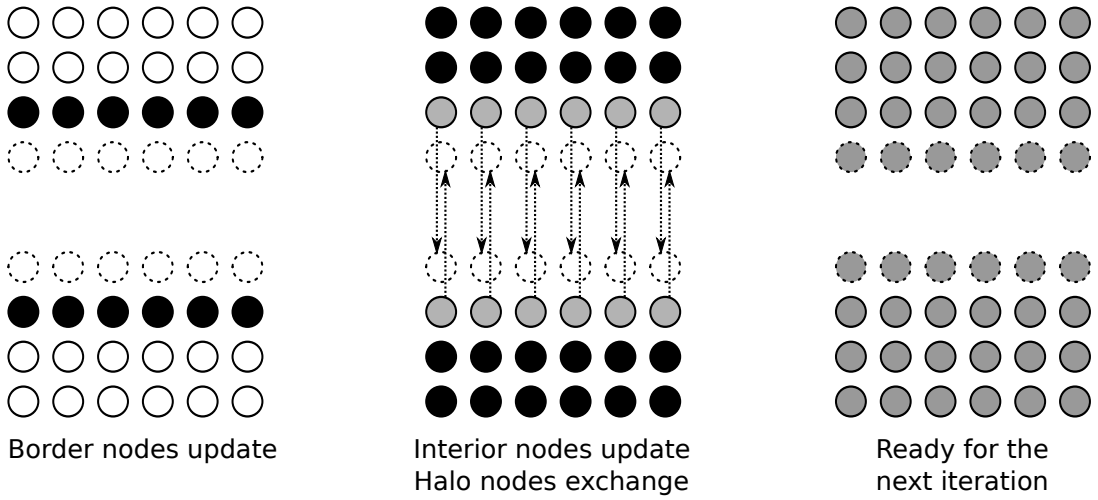


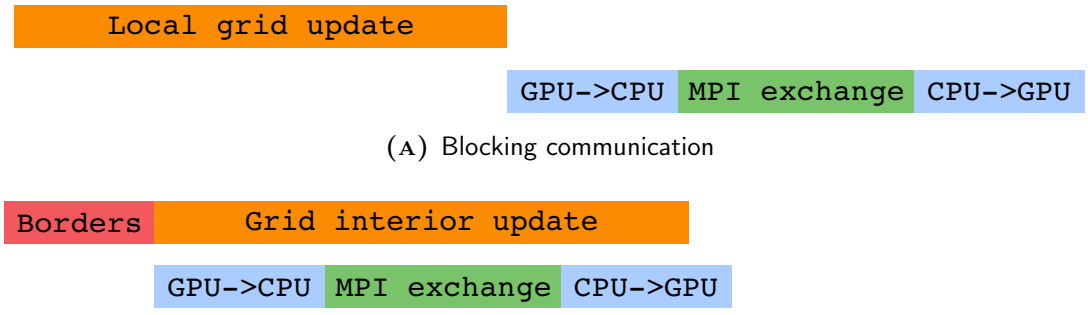
Figure 3.2: Overlapping the grid update and the halo nodes exchange in the non-blocking Hybrid Jacobi Solver (6×6 grid on two GPUs). Legend: white — nodes not yet processed, black — nodes updated in parallel, grey — already processed nodes, dotted — halo nodes, dotted lines show the three-stage data transfers (GPU→CPU, MPI, CPU→GPU).

executed with thread blocks of 32×8 threads. The initial experiments have confirmed that this setup results in the shortest execution time, except for the Surface Memory Kernel, for which blocks of 16×16 threads were optimal.

3.2.1 Kernel performance comparison

The CUDA Visual Profiler was used to assess the fraction of coalesced reads on different devices. Devices with Compute Capabilities 1.3 and 2.0 aim at minimising the number of 32-, 64-, and 128-bytes reads. Since in the case of the Jacobi solver it is impossible to meet the alignment requirement for all reads, some excess data is loaded. Initial experiments indicated that approximately 40% of data is read in excess, limiting the effective memory bandwidth to 60% of the peak performance.

On Tesla C1060 (Compute Capability 1.3), the performance of all three kernels increases with increasing grid size and stabilises at $n = 768$ (Figure 3.4A). The Global Memory Kernel yields the worst performance, due to lack of any caching mechanism. Thanks to the manual caching implemented in the Shared Memory



(A) Blocking communication

(B) Non-blocking communication

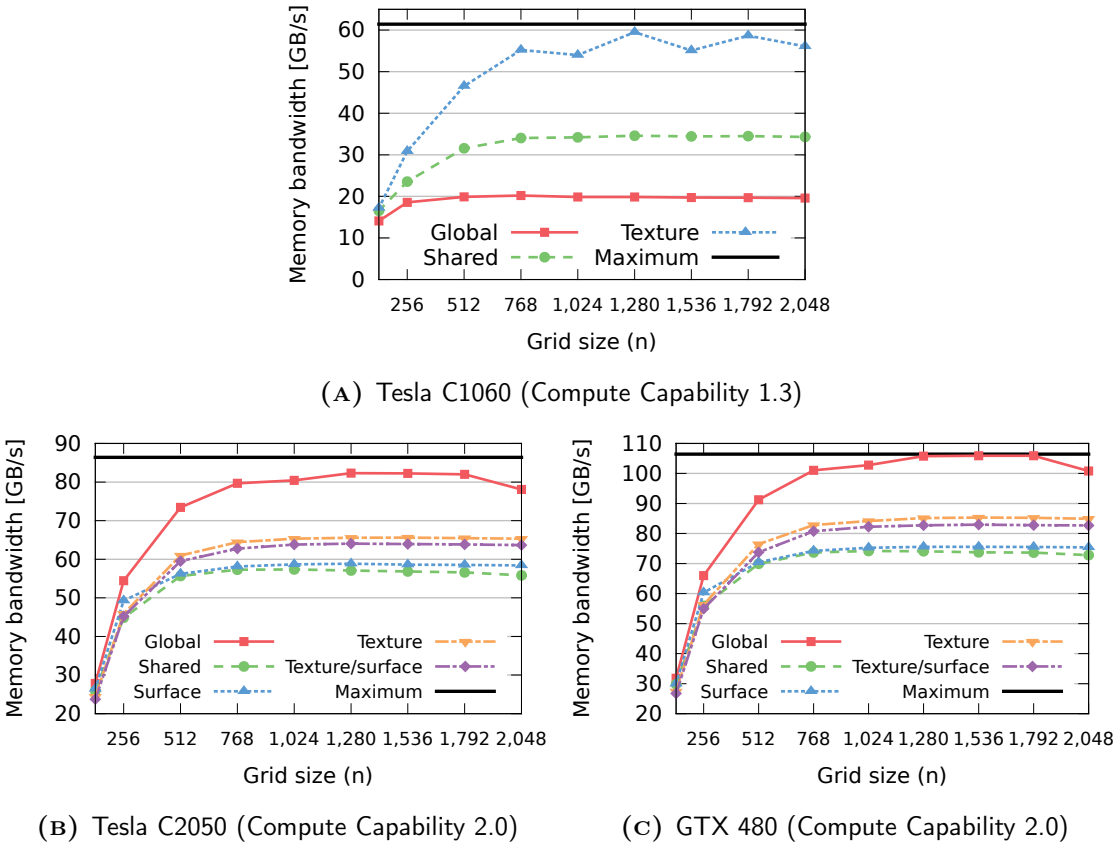
FIGURE 3.3: Hybrid Jacobi Solver timelines in different communication models. The non-blocking communication (B) allows us to overlap computation and data transfers.

Kernel, memory throughput is higher, however only roughly two times, instead of four. This is connected with the unbalanced workload within the blocks — border threads have to read two values from the global memory, while the inner threads read only one value. The Texture Memory Kernel introduces automatic caching, which allows us to obtain the best performance on devices with Compute Capability 1.3. The best attainable memory throughput is at roughly 60% of peak bandwidth (61.44 GB/s). Performance close to that analytical maximum can be observed for the Texture Memory Kernel on large grids ($n \geq 768$).

The performance of the Global Memory Kernel changes significantly on devices with Compute Capability 2.0, due to the introduction of L1 and L2 caches (Figures 3.4B and 3.4C). The CUDA Toolkit allows us to set the L1 cache size to either 16 or 48 KB (NVIDIA, 2011). Increasing that size to 48 KB, resulted in a further 5% increase in performance, confirming the positive impact of automatic caching.

Similar to devices with Compute Capability 1.3, the performance of all kernels increases with increasing grid size and stabilises at $n = 768$. Moreover, the maximum effective memory throughput (86.4 GB/s and 106.44 GB/s on Tesla C2050 and GTX 480, respectively) is at 60% of the peak bandwidth due to some unaligned global memory reads. On both devices with Compute Capability 2.0, the Global Memory Kernel show performance close to the analytical maximum ($n \geq 768$).

The Texture Memory Kernel also show consistently good performance — the observed memory bandwidth is at roughly 80% of that of the Global Memory Kernel.



(A) Tesla C1060 (Compute Capability 1.3)

(B) Tesla C2050 (Compute Capability 2.0)

(C) GTX 480 (Compute Capability 2.0)

FIGURE 3.4: Performance of five-point stencil kernels for the Hybrid Jacobi Solver for 2D Laplace's Equation on devices with different Compute Capabilities. As kernels are memory-bound, performance is expressed in terms of memory bandwidth.

The worst performance was observed for the Shared Memory Kernel, again due to the unbalanced workload in each block of threads.

Surface memory is not well documented and it was difficult to predict its performance, but experiments showed that the Surface Memory Kernel obtains bandwidth similar to that of the Shared Memory Kernel. The Texture and Surface Memory Kernel yielded performance consistently slightly lower than the Texture Memory Kernel due to slower write operations.

To summarise, the Texture Memory Kernel offers the best performance on devices with Compute Capability 1.3, and the Global Memory Kernel on devices with Compute Capability 2.0. Both kernels obtain the maximum effective memory bandwidth

for sufficiently large grids, on respective devices. This was achieved thanks to efficient use of the implicit caching mechanism.

3.2.2 Hybrid Solver on Multiple GPUs

The best performing kernels were integrated into the Hybrid Jacobi Solver for 2D Laplace's Equation. Experiments were run using up to three identical GPUs using three media for interconnect: all GPUs sharing the same motherboard (Tesla C1060), and GPUs on three separate machines (Tesla C2050) connected with Fast Ethernet (100 Mbps) and Gigabit Ethernet (1 Gbps).

Figure 3.5 shows the solver execution time breakdown with respect to grid update, and CPU-GPU and MPI transfers. In the case of non-blocking communication, the higher of two bars determines the overall execution time. As expected, the proportion of time required for computing is increasing significantly with the grid size ($\mathcal{O}(n^2)$ work complexity in comparison to $\mathcal{O}(n)$ complexity of communication step), and decreasing with increasing GPU count (smaller stripe on each GPU). Computation times in Ethernet-based systems are the same, but lower than on shared memory system (Tesla C1060 has roughly 30% lower peak memory bandwidth).

In Ethernet-based systems, MPI transfers take a substantial amount of time. This is especially striking in the case of 100 Mbps interconnect, where MPI transfers are clearly dominating over computation. In the case of Gigabit connectivity, the proportion of MPI transfers is still significant, but comparable to computation fraction. When all GPUs are sharing the same motherboard, MPI transfers are carried out via much faster shared memory and their proportion in the overall execution time is small. Regardless of connectivity, the proportion of MPI transfers is decreasing with increasing grid size (amount of transferred data grows in linear fashion).

Since the data chunks exchanged in each iteration are relatively small, the CPU-GPU transfer setup is determining its execution time. Indeed, the total time spent on these transfers is similar regardless of the architecture and the grid size, hence their proportion is decreasing with the increasing grid size. There is one exception though: on devices with Compute Capability 1.x the setup overhead for asynchronous transfer is significantly higher than for synchronous communication (Lawlor, 2009), which explains higher times in the non-blocking case on shared memory system.

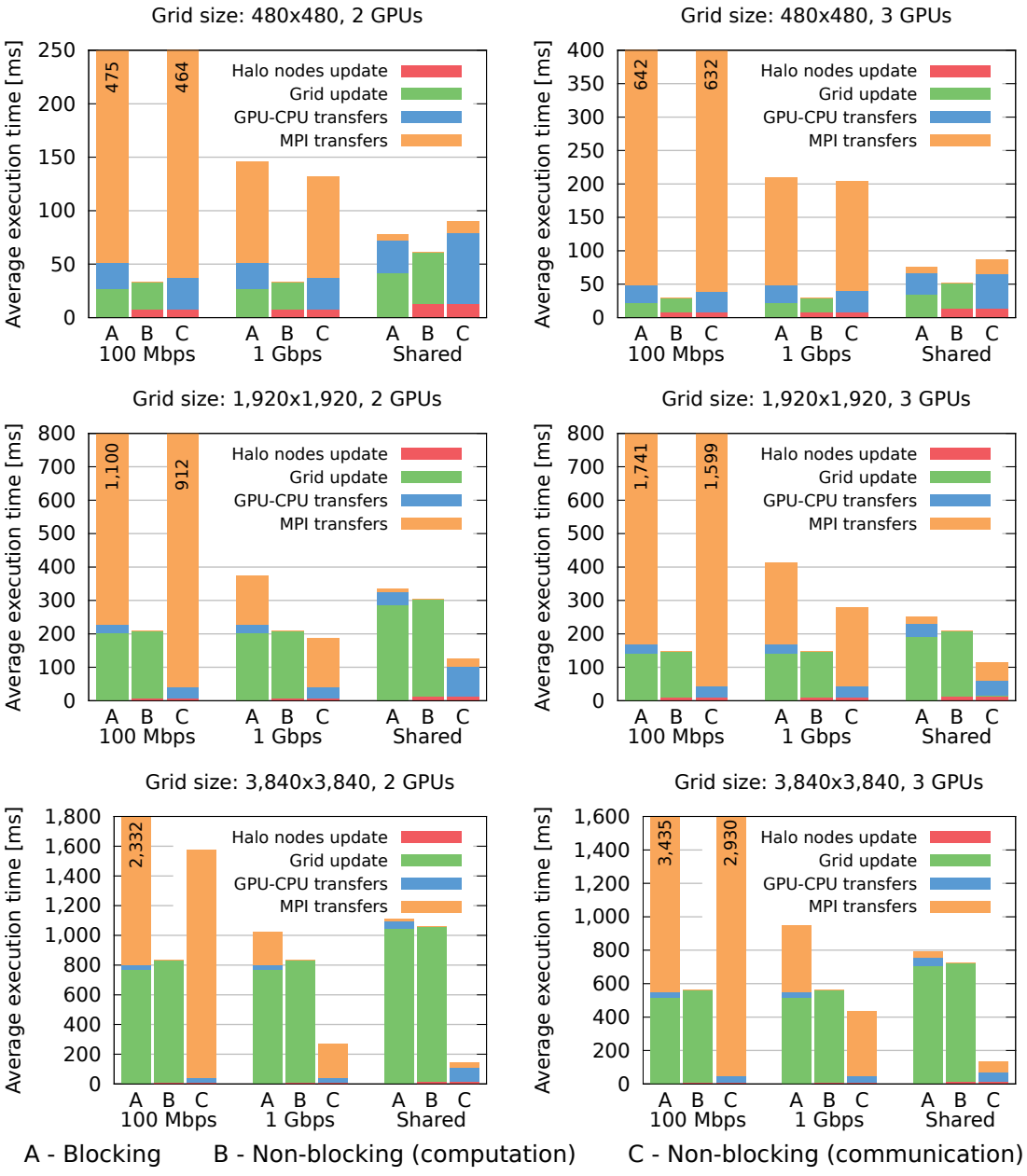


Figure 3.5: The execution time breakdown in the Hybrid Jacobi Solver for 2D Laplace's Equation. In the case of the non-blocking solver, the time is determined by the higher of B and C bars.

3.2.3 Benefits of Non-blocking Communication

The results presented above lead to a conclusion that the use of non-blocking memory transfers always yields better performance than blocking communication. If either computation or communication dominates greatly, then the benefits are negligible, however that is often not the case.

Figure 3.6 compares the time required for computation and communication, depending on the interconnect, and the grid size. The intersection point indicates the grid size for which computation and communication times are equal, hence these operations can fully overlap. In that case, a non-blocking solver is two times faster than its blocking counterpart. For n values close to that optimal point, the benefits from non-blocking communication are the most significant.

In the case of shared memory system, the intersection point is at a relatively low $n < 960$ value. Here, computation quickly becomes dominant over communication and benefits from overlapping are relatively small. On the other hand, in the Fast Ethernet system the intersection point is at some large $n > 4{,}800$ value, hence communication is dominant. Finally, in the system with Gigabit Ethernet the time required for computation and communication is similar, hence the benefits from overlapping are relatively high.

3.2.4 Communication Bottleneck

The time required for MPI transfers had the greatest impact in the system with the 100 Mbps network. Indeed, the observed network utilisation was close to the theoretical maximum even for small grids (Figure 3.7). Similarly, memory bandwidth achieved in Gigabit Ethernet on large grids constituted over 65% of the theoretical maximum. However, in this case, MPI transfers were no longer the limiting factor for the entire hybrid solver. Finally, transfer rates observed in the system with all GPUs sharing one motherboard were an order of magnitude higher than those in the Gigabit Ethernet case.

When three GPUs were used, the amount of data transferred by each node was doubled. This helped to achieve transfer rates slightly, but consistently, higher than in the two GPUs case. This is not as consistent in the shared memory system, due

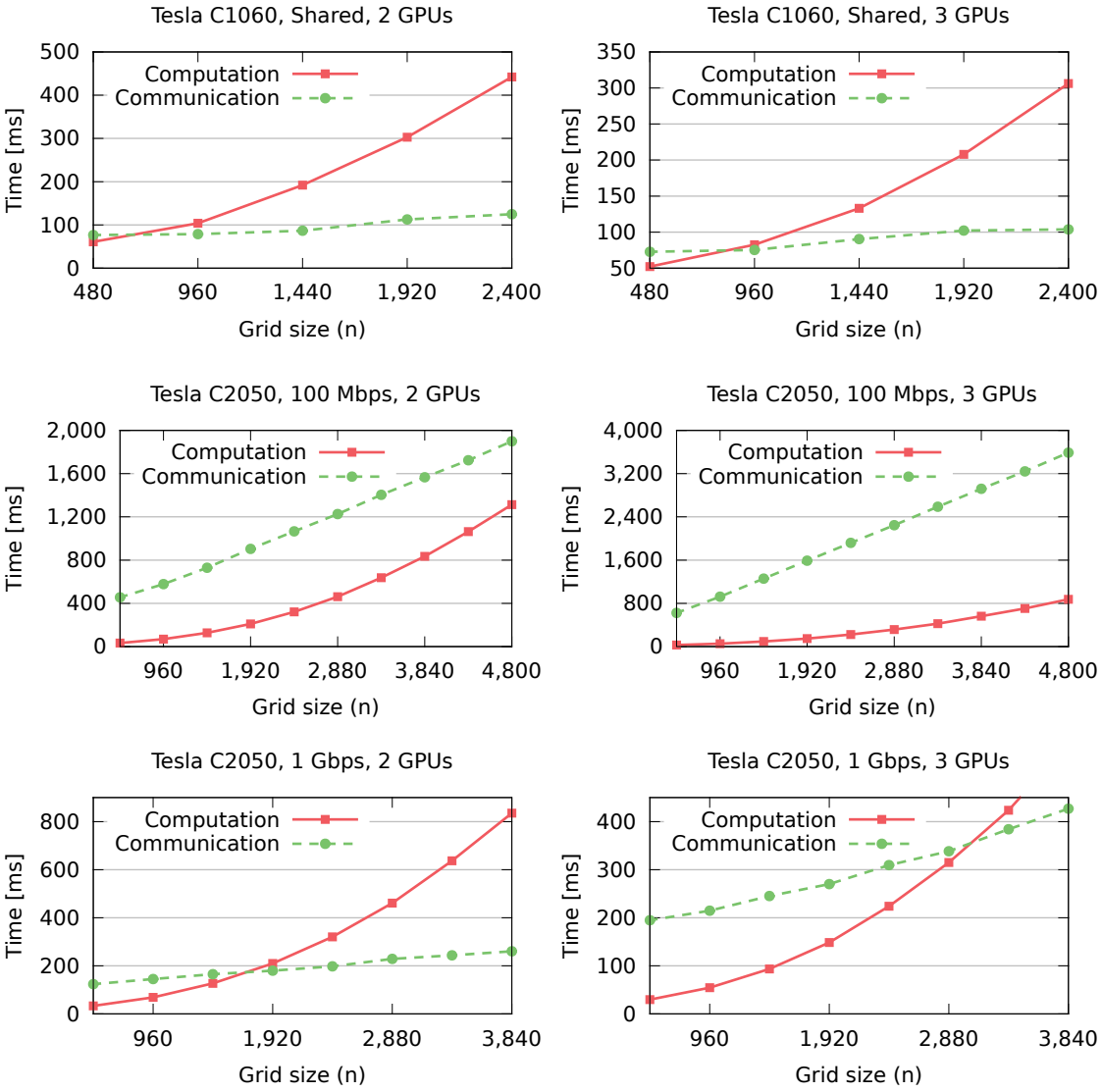


Figure 3.6: Computation and communication time in the non-blocking solver. These operations can overlap, therefore the one that takes longer determines the overall execution time. For n values close to the intersection point, the benefits from non-blocking communication are the most significant.

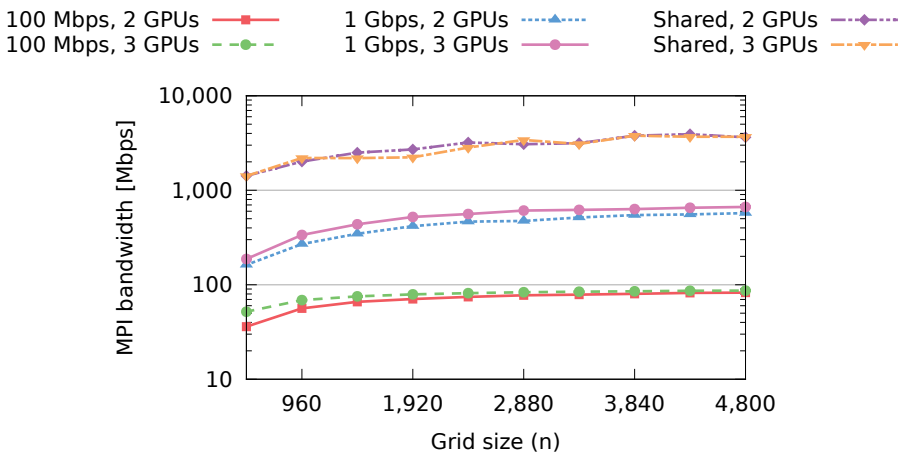


Figure 3.7: Throughput of MPI transfers in systems with different connectivity.

to short transfer times and lower measurement accuracy — in this case, the relative standard deviation of results was higher than 5%.

To conclude, the results presented in this, and previous subsections, indicate that the interconnect between the nodes may have a crucial impact on the hybrid solver performance. The smallest impact on performance is observed in case of the shared memory system, but its scalability is limited by the maximum number of GPUs connected to the same motherboard.

Scalability is not a problem in distributed memory systems, but in the case of a Fast Ethernet connectivity, MPI transfers become a bottleneck compromising the solver performance. When a relatively cheap Gigabit Ethernet is used, MPI transfers still pose a significant fraction of execution time, but can be effectively overlapped with computation. The results presented in this chapter indicate that using an Infiniband network would allow for even better scalability, but at much higher installation costs. This observation is consistent with the results presented by Micikevicius (2009) and White and Dongarra (2011).

3.2.5 Parallel Efficiency of the Hybrid Solver

When a Fast Ethernet network is used for connectivity, parallel efficiency of the hybrid solver is low (Figure 3.8). The solver run on multiple GPUs is actually slower than a single GPU configuration — this is the case whenever the efficiency

is below 50% (2 GPUs) or 33% (3 GPUs). Parallel efficiency of the solver running on two GPUs is roughly two times higher. This is connected with the fact that the amount of data transferred by each node is doubled in the three GPUs case.

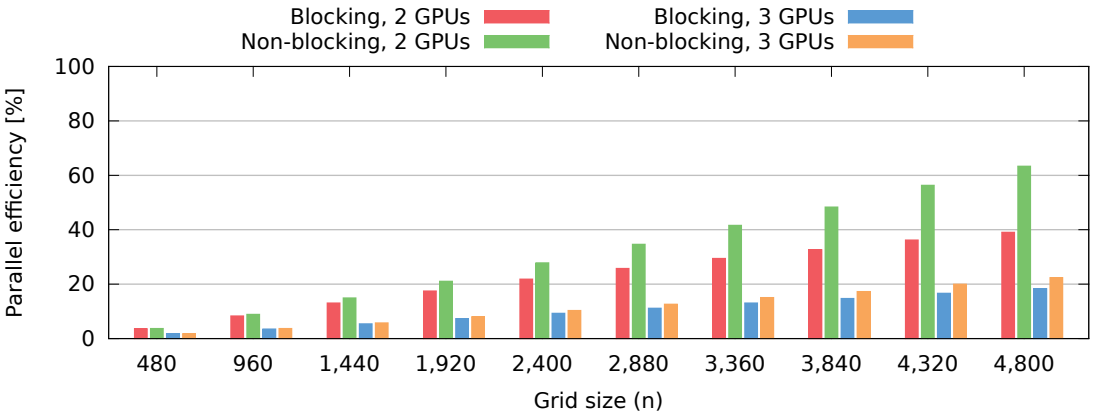


Figure 3.8: Parallel efficiency of the Hybrid Jacobi Solver with Fast Ethernet (100 Mbps) connectivity. Due to communication overheads a single GPU configuration is faster than the multi-GPU solver. The efficiency on 3 GPUs is two times lower due to a doubled amount of data transferred by the GPU handling the middle data stripe.

Parallel efficiency is significantly better when a Gigabit Ethernet interconnect is used instead (Figure 3.9). In this case, the multi-GPU solver is faster than a single GPU configuration when $n \geq 1{,}440$. Noticeably higher efficiency in the case of non-blocking communication is connected with relatively high speed-up from overlapping. For large grids ($n \geq 3{,}360$), parallel efficiency is consistently above 90%, regardless of the number of GPUs. This confirms that the proposed hybrid solver is capable of almost linear speed-ups with increasing number of GPUs, for sufficiently large grids (good weak scalability).

Finally, when all GPUs are installed on the same motherboard and MPI transfers are carried out via shared memory, the time required for communication is small in comparison to computation except for small grids (Figure 3.10). Parallel efficiency is high regardless of the communication model used, but consistently higher (and close to 100%) for the non-blocking communication. Therefore, the scalability is even better than in the Gigabit Ethernet case, but is limited by the maximum number of GPUs connected to the same motherboard (typically four).

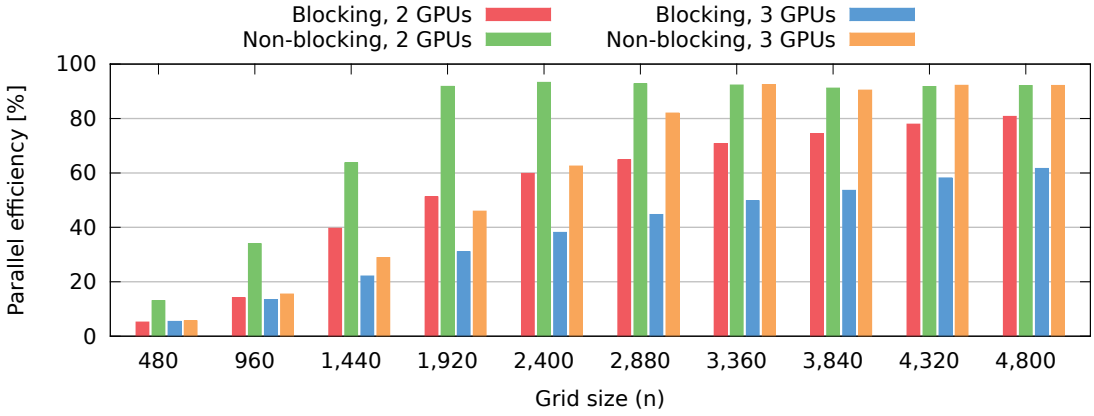


Figure 3.9: Parallel efficiency of the Hybrid Jacobi Solver with Gigabit Ethernet (1 Gbps) connectivity. The non-blocking solver offers significantly better performance and is capable of achieving close to linear speed-up for sufficiently large grids.

3.2.6 Dynamic Load Balancing in Heterogeneous Architectures

The Hybrid Jacobi Parallel Solver for 2D Laplace's Equation is not limited to work on homogeneous computing resources. In fact, any CUDA capable device can be used, as well as multi-core CPUs. In the former case, it is advisable to use devices with at least Compute Capability 1.3 — performance of older devices is greatly limited by strict coalescing requirements and typically much lower peak memory bandwidth. In the the multi-core CPU case, CPU-GPU transfers are not required, enabling good performance, especially for smaller grids requiring less computation.

In a heterogeneous environment, dividing the grid into stripes with equal heights would result in an unbalanced workload and faster GPUs waiting idle for the slower ones to complete updating their stripe. The overall performance would be limited by the slowest device.

As was shown in Subsection 3.2.1, for sufficiently large grids our GPU kernels are able to obtain 60% of peak memory bandwidth on devices with Compute Capability 1.3 or higher. This information and device peak memory bandwidth available through CUDA API can be used to balance the workload, i.e. each device would get a stripe proportional to its peak memory bandwidth.

Results for the system consisting of three different GPUs (Tesla C1060, Tesla C2050, GTX 480) and Gigabit Ethernet interconnect are presented in Figure 3.11. In the

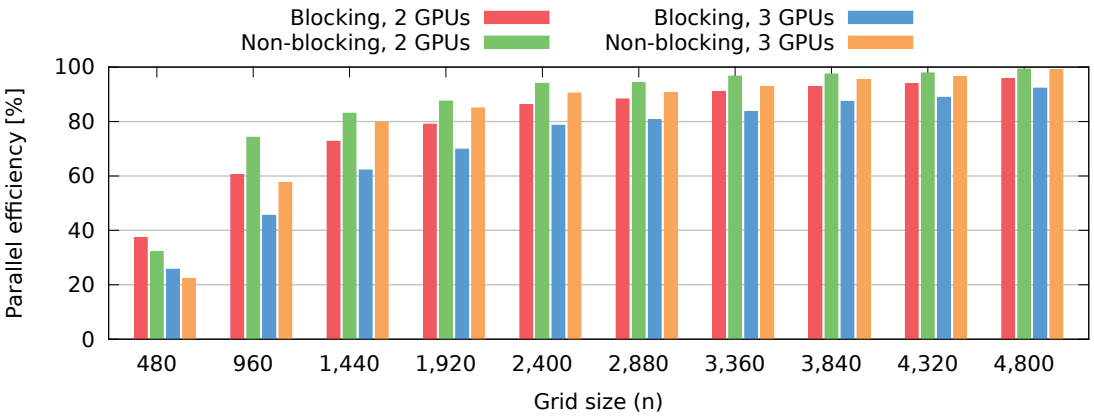


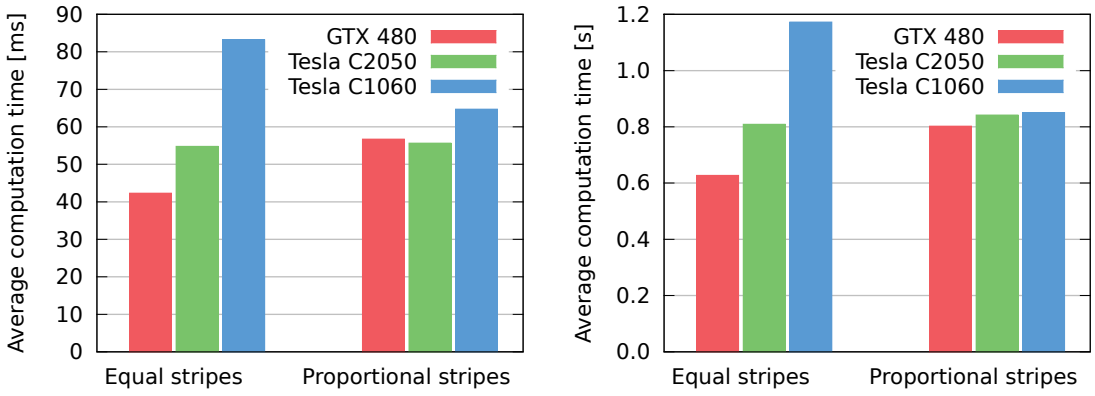
Figure 3.10: Parallel efficiency of the Hybrid Jacobi Solver with shared memory connectivity. In this case the communication overhead is low and high efficiency is achieved by blocking and non-blocking solvers even for small grids. However, the non-blocking solver is consistently faster, especially on the 3 GPUs configuration, where more data is transferred.

case of equal height stripes, the Tesla C1060 requires roughly two times more time than GTX 480 and 30% more time than Tesla C2050, regardless of the grid size.

In the case of a 1,200 × 1,200 grid, using stripes with proportional height reduces the spread between the fastest and the slowest GPU to roughly 14%, and decreases overall computation time by 23%. In the case of a larger 4,800 × 4,800 grid, the performance gain from the load balancing is even higher: the spread is reduced to below 6%, and the computation time is reduced by almost 28%. As all devices are busy for most of the time, there is little space for further improvement.

3.3 Conclusions

- This chapter presents a range of techniques aiming at reducing communication overhead in hybrid CPU-GPU parallel architectures. These methods have been applied in the implementation of the Hybrid Jacobi Solver for 2D Laplace's Equation. This problem was chosen because of its theoretical importance, and low compute-to-communication ratio, which makes efficient parallelisation on GPU platforms difficult.



(A) Grid size: 1,200 × 1,200 (B) Grid size: 4,800 × 4,800

FIGURE 3.11: Impact of the automatic load balancing on the execution time on heterogeneous platforms. Dividing the grid into stripes proportional to the GPU peak performance results in almost perfectly balanced workload.

- Five CUDA kernels utilising different GPU memory spaces have been implemented and compared. On devices with Compute Capability 1.3, the use of texture memory resulted in the best performance, whereas on devices with Compute Capability 2.0 the global memory implementation was the fastest, due to their implicit caching mechanism. Regardless of Compute Capability, the fastest kernel on each device reached the analytical maximum for memory bandwidth (up to 106 GB/s on GTX 480).

- Due to asynchronous memory transfer mechanisms in MPI and CUDA APIs, it is possible to overlap computation and communication. The proposed hybrid solver using non-blocking communication is up to two times faster than its blocking counterpart.

- When a relatively slow Fast Ethernet interconnect is used, the time required for communication between graphics cards becomes dominant, resulting in increasing execution time when more GPUs are used. This makes multi-GPU systems with Fast Ethernet interconnect unsuitable for this application.

- When all GPUs are connected to the same motherboard, even for small grids all communication can overlap with computation, resulting in linear speed-ups. However, this architecture is limited by the maximum number of GPUs

that can be connected to a single motherboard (typically four).

- Finally, the use of Gigabit Ethernet can help to alleviate both problems. Inter-GPU communication still takes a significant proportion of overall execution time, but it can be completely overlapped with computation for sufficiently large grids, yielding parallel efficiency close to 100%. Since the number of GPUs is no longer limited, this architecture provides good scalability. The performance can be further improved by using the Infiniband network, but at a significant installation cost.

- The proposed load balancing technique for heterogeneous multi-GPU environments allows us to balance the workload almost perfectly — the fastest device is idle for at most 6% of the time.

Chapter 4

Case Study: Hybrid SOR Solver for 2D Poisson's Equation

The previous chapter focused on communication overheads in hybrid CPU-GPU systems and on techniques to alleviate their impact on the speed of parallel algorithms. Data dependencies, operations ordering, and data layout can also significantly affect the performance on parallel platforms. The case study presented in this chapter aims to investigate the effect of these factors in the Successive Over-Relaxation (SOR) solver for 2D Poisson's Equation.

The choice of the model problem was motivated by several considerations. First, the SOR method involves non-trivial data dependencies, that complicate efficient parallelisation. Techniques for overcoming this problem are well-known and are discussed in this chapter. Secondly, the SOR method has been studied for over 60 years and its theoretical properties and performance characteristics are well understood. Finally, when applied to the solution of Poisson's Equation, SOR exhibits good performance, comparable to more sophisticated methods, e.g. Conjugate Gradients.

The introduction to Successive Over-Relaxation method, the model PDE problem, and related work are presented in Section 4.1. Section 4.2 investigates the complexity and the impact of techniques enabling efficient parallelisation on the SOR method convergence. The details of all implementations considered in this study, are given in Section 4.3, followed by the discussion of numerical experiments results in Section 4.4. Finally, the conclusions are given in Section 4.5.

4.1 Introduction

Successive Over-Relaxation was first proposed in the PhD dissertation by Young (1950). It is classified as a stationary iterative method, similar to Jacobi and Gauss-Seidel methods. The Gauss-Seidel approach is a modification of the Jacobi method that improves the rate of convergence by a factor of two. SOR is a generalisation of the Gauss-Seidel method introducing a relaxation parameter ω.

When the optimal ω value is used, the rate of convergence is improved by an order of magnitude in comparison to the Gauss-Seidel method (SOR with $\omega = 1$). However, the optimal ω value depends on the linear system solved, and in most cases it is impractical to compute it. In consequence, various heuristics have been studied, e.g. for linear systems coming from discretisation of PDEs, $\omega = 2 - \mathcal{O}(h)$ can be used, where h is the mesh spacing (Barrett et al., 1994, p. 10).

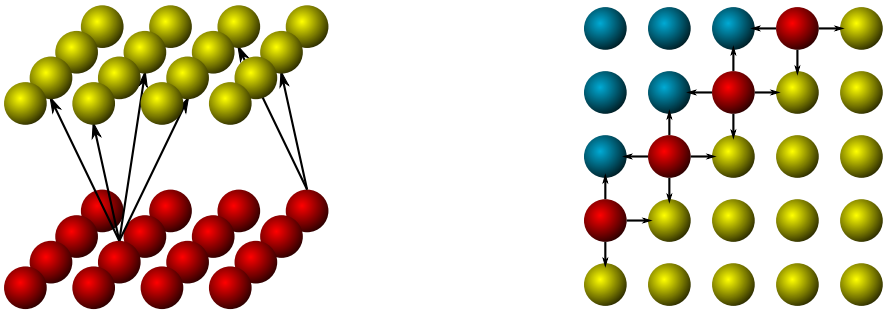
In case of the 2D Poisson's Equation, the optimal relaxation parameter is explicitly known (Demmel, 1997, §6.5.4):

$$\omega_{opt} = \frac{2}{1 + \sin \pi h}. \tag{4.1}$$

Furthermore, this result holds in an arbitrary number of dimensions of Poisson's Equation. The proof is provided by Yang and Gobbert (2009).

In the Jacobi method, new values depend solely on old values computed in the previous iteration (Figure 4.1 A). In consequence, the task of updating all grid values is embarrassingly parallel. In contrast, in the classic SOR method, each new value also depends on the values computed in the same iteration (Figure 4.1 B). In this case, only values on consecutive diagonals can be computed in parallel.

To alleviate this limitation, the *Red-Black ordering* was introduced to improve parallel processing performance (Adams and Ortega, 1982). In this scheme, the grid elements are partitioned, so that the adjacent nodes have different colours. In the case of Poisson's Equation, two colours are sufficient. In consequence, there are no data dependencies between the nodes with the same colour, and these can be updated in parallel. The Red-Black SOR iteration consists of updating the red nodes first, followed by the black nodes update (Figure 4.2).



(A) Jacobi method (B) Classic SOR method

FIGURE 4.1: Data dependencies in Jacobi and SOR methods (indicated by arrows). Jacobi method requires two copies of grid nodes. Colour legend: yellow — old values, blue — already updated nodes, red — nodes that can be updated in parallel.

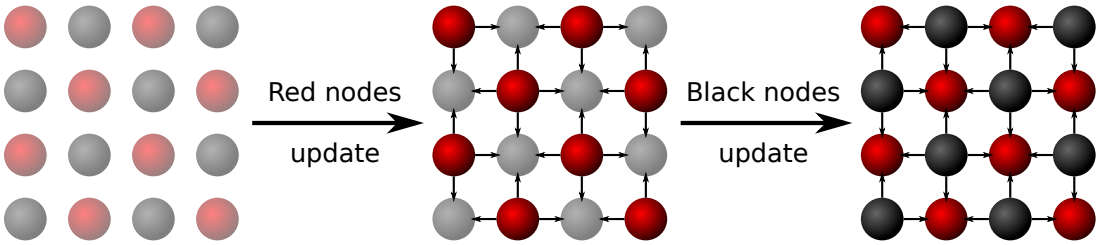


FIGURE 4.2: Iteration in SOR with Red-Black colouring. The transparent nodes indicate the old values, small arrows show the data dependencies.

Another consequence of using the Red-Black scheme, is that the order in which the grid nodes are updated is changed. In general, changing the order in which the grid elements are updated affects the convergence rate. In some cases, the SOR method may even break down and fail to converge. However, the Red-Black scheme is an example of *consistent ordering* and is guaranteed to converge at the same rate asymptotically as the classic SOR method (Young, 1971).

The Red-Black colouring scheme became the state-of-the-art solver for Poisson's Equation, both in the 2D case (e.g. Evans, 1984) and in the 3D case (e.g. Brown et al., 1993). The solvers proposed in this study also follow this scheme, although Di et al. (2012) recently proposed a Multi-layer SOR implementation designed for multiple GPUs. Their approach is discussed in more detail in Subsection 4.4.6.

4.1.1 Model Problem

The general form of the 2D Poisson's Equation is defined as

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f \tag{4.2}$$

on some domain Ω. In this study, rectangular domains are considered, i.e.

$$\Omega = \{(x, y) : x_{min} < x < x_{max},\ y_{min} < y < y_{max}\}. \tag{4.3}$$

Discretising Equation (4.2) on domain (4.3) with the finite difference method using a uniform $n \times n$ grid, leads to the following set of linear equations:

$$-v_{i-1,j} - v_{i,j-1} + 4v_{i,j} - v_{i,j+1} - v_{i+1,j} = h^2 f_{i,j} \tag{4.4}$$

$$\text{assuming uniform grid spacing } h = \frac{x_{max} - x_{min}}{n+1} = \frac{y_{max} - y_{min}}{n+1}, \text{ and}$$

$$v_{i,j} \approx \phi(x_i, y_j), \quad f_{i,j} = f(x_i, y_j) \quad \text{where } x_i = x_{min} + ih, \quad y_j = y_{min} + jh.$$

Our experiments were conducted on the following model 2D Poisson problem with homogeneous Dirichlet boundary conditions:

$$\begin{aligned} \Omega &= \{(x, y) : -1 < x,\ y < 1\}, \\ \nabla^2 \phi &= -2\pi^2 \sin(\pi x) \sin(\pi y) \text{ in } \Omega, \\ \phi &= 0 \text{ on } \partial\Omega. \end{aligned}$$

The solution to this problem is presented in Figure C.1 A in Appendix C.

4.1.2 Successive Over-Relaxation

A system of n linear equations can be expressed in the matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{4.5}$$

The matrix $\mathbf{A}$ can be then decomposed into a diagonal component $\mathbf{D}$, and strictly lower and upper triangular components $\mathbf{L}$ and $\mathbf{U}$:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}. \tag{4.6}$$

By substituting $\mathbf{A}$, according to Equation (4.6), Equation (4.5) can be rewritten as:

$$(\mathbf{D} + \mathbf{L} + \mathbf{U})\mathbf{x} = \mathbf{b}, \tag{4.7}$$

$$\omega(\mathbf{D} + \mathbf{L} + \mathbf{U})\mathbf{x} = \omega\mathbf{b}, \tag{4.8}$$

$$(\mathbf{D} + \omega\mathbf{L})\mathbf{x} = \omega\mathbf{b} - [\omega\mathbf{U} + (\omega - 1)\mathbf{D}]\,\mathbf{x}. \tag{4.9}$$

In each iteration of the SOR method, the linear system (4.9) is solved to find $\mathbf{x}^{(r+1)}$, using the already known vector $\mathbf{x}^{(r)}$ to evaluate the right-hand side. In consequence, vector $\mathbf{x}^{(r+1)}$ can be expressed in the matrix form as:

$$\mathbf{x}^{(r+1)} = (\mathbf{D} + \omega\mathbf{L})^{-1}\left(\omega\mathbf{b} - [\omega\mathbf{U} + (\omega - 1)\mathbf{D}]\,\mathbf{x}^{(r)}\right) = \mathbf{M}_\omega\mathbf{x}^{(r)} + \mathbf{c}. \tag{4.10}$$

Since $(\mathbf{D} + \omega\mathbf{L})$ is lower triangular, elements of vector $\mathbf{x}^{(r+1)}$ can be computed using forward substitution:

$$x_i^{(r+1)} = (1 - \omega)x_i^{(r)} + \frac{\omega}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(r+1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(r)}\right). \tag{4.11}$$

In the case of the 2D Poisson's Equation on an $n \times n$ grid, the linear system consists of n^2 equations. The corresponding matrix $\mathbf{A}$ has at most five non-zero coefficients in each row: value 4 on the diagonal, and up to four -1 values, according to Equations (4.4). In consequence, the SOR iteration expressed in Equation (4.11) simplifies to:

$$v_{i,j}^{(r+1)} = (1 - \omega)v_{i,j}^{(r)} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j}^{(r+1)} + v_{i,j-1}^{(r+1)} + v_{i,j+1}^{(r)} + v_{i+1,j}^{(r)}\right). \tag{4.12}$$

The pseudo-code for the Successive Over-Relaxation method for the 2D Poisson's Equation is provided in Algorithm 4.1.

4.1.3 Parallel Successive Over-Relaxation

In order to achieve an acceptable level of parallelism, the Red-Black ordering has to be used — grid nodes with the same colour can be updated concurrently. The parallelisation is straightforward on shared memory architectures: first the red nodes are updated, then the black ones. The only parallelisation overhead is the need to

Algorithm 4.1 Successive Over-Relaxation for 2D Poisson's Equation

while Stopping criteria are not met **do**
  for i = 1 **to** n **do**
    for j = 1 **to** n **do**
      $v_{i,j} = (1-\omega)v_{i,j} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} + v_{i+1,j}\right)$
    end for
  end for
end while

Algorithm 4.2 Multi-threaded parallel SOR (shared memory)

while Stopping criteria are not met **do**
  for all $v_{i,j}$ nodes that are red (one thread per value) **do**
    $v_{i,j} = (1-\omega)v_{i,j} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} + v_{i+1,j}\right)$
  end for
  Synchronise threads
  for all $v_{i,j}$ nodes that are black (one thread per value) **do**
    $v_{i,j} = (1-\omega)v_{i,j} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} + v_{i+1,j}\right)$
  end for
  Synchronise threads
end while

synchronise all the threads after each colour is updated, i.e. twice per iteration (Algorithm 4.2) instead of one (Algorithm 4.1).

The situation becomes more complicated in distributed memory systems. The grid is decomposed into equally sized subdomains. In this study, the subdomains form horizontal stripes. To ensure correct computation of the border nodes, each processing element must hold an up-to-date copy of *halo nodes* — the border nodes from the two adjacent subdomains. These have to be exchanged after each colour is updated, i.e. two times per iteration (Algorithm 4.3). Finally, once all the iterations are completed, the subdomains have to be gathered to form a final solution. Obviously, these communication operations impose an additional parallelisation overhead.

It is tempting to omit the halo node exchange between updates of red and black nodes, as it would be possible in the case of parallel Jacobi solver. However, as will be shown in Section 4.2, this may have an adverse effect on algorithm convergence.

Algorithm 4.3 Parallel SOR in a distributed memory system using MPI

while Stopping criteria are not met **do**
  for $v_{i,j}$ nodes in the local stripe that are red **do**
    $v_{i,j} = (1-\omega)v_{i,j} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} + v_{i+1,j}\right)$
  end for
  Exchange red halo nodes (`MPI_Send`, `MPI_Recv`)
  for $v_{i,j}$ nodes in the local stripe that are black **do**
    $v_{i,j} = (1-\omega)v_{i,j} + \frac{\omega}{4}\left(h^2 f_{i,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} + v_{i+1,j}\right)$
  end for
  Exchange black halo nodes (`MPI_Send`, `MPI_Recv`)
end while
Combine all local stripes to form a final solution (`MPI_Gather`)

4.2 Convergence and Complexity Analysis

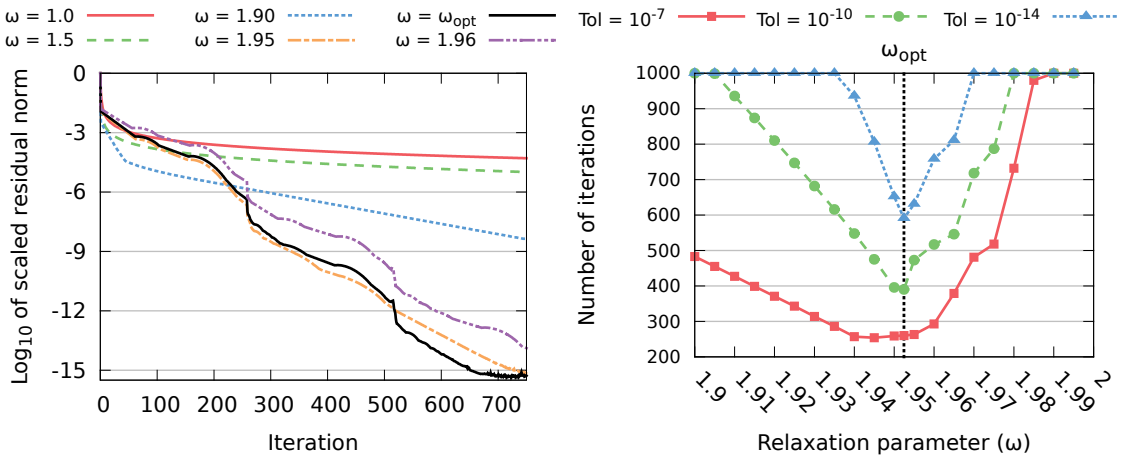
This section investigates the convergence rate of SOR with different relaxation parameter values and orderings. All the experiments were conducted on the model problem presented in Subsection 4.1.1, using a 128×128 grid and double precision arithmetic. The quality of the solution vector was measured with *scaled residual norms*, computed according to the formula proposed by Barrett et al. (1994, Ch. 4):

$$res = \frac{||\mathbf{r}||}{||\mathbf{A}|| \cdot ||\mathbf{x}|| + ||\mathbf{b}||}\,. \tag{4.13}$$

According to Equation (4.1), $\omega_{opt} = 2/(1 + \sin\frac{\pi}{129}) \approx 1.9524558$ is the optimal relaxation parameter when $n = 128$ — indeed, for this value the observed convergence was the fastest. Moreover, even a relatively small change may result in a significantly worse convergence rate (Figure 4.3). Increasing the relaxation parameter by just 0.0075 resulted in a noticeable change in convergence, while a decrease by 0.05 led to a significant convergence rate deterioration (Figure 4.3 A).

In the case of the Jacobi solver, the rate of convergence is (Demmel, 1997, §6.5.4)

$$\rho_J = \cos\frac{\pi}{n+1} \approx 1 - \frac{\pi^2}{2(n+1)^2} = 1 - \mathcal{O}(h^2). \tag{4.14}$$



(A) Changing relaxation parameter

(B) Iterations to convergence

FIGURE 4.3: SOR convergence rate for changing relaxation parameter (128×128 grid, double precision). Optimal relaxation parameter $\omega_{opt} = 1.9524558$. The result of 1,000 iterations in (B) means that the method did not converge to the specified error tolerance. A significantly improved convergence rate, $\rho_S = 1 - \mathcal{O}(h)$, is only observed in the small range around the optimal ω. Outside this range, the rate of convergence gradually deteriorates to that of the Gauss-Seidel method, which is an order of magnitude slower, i.e. $\rho_{GS} = 1 - \mathcal{O}(h^2)$.

As a consequence, the number of iterations R, required to decrease the error by a factor of $e^{-1} = \exp(-1)$ is the solution to the equation

$$\left(1 - \frac{\pi^2}{2(n+1)^2}\right)^R = e^{-1}, \tag{4.15}$$

and for sufficiently large n values can be estimated as

$$R \approx \frac{2(n+1)^2}{\pi^2} = \mathcal{O}(n^2). \tag{4.16}$$

Moreover, each Jacobi iteration requires $\mathcal{O}(n^2)$ time, and in consequence, the Jacobi method has $\mathcal{O}(n^4) = \mathcal{O}(N^2)$ time complexity, where $N = n^2$ is the number of elements in the grid.

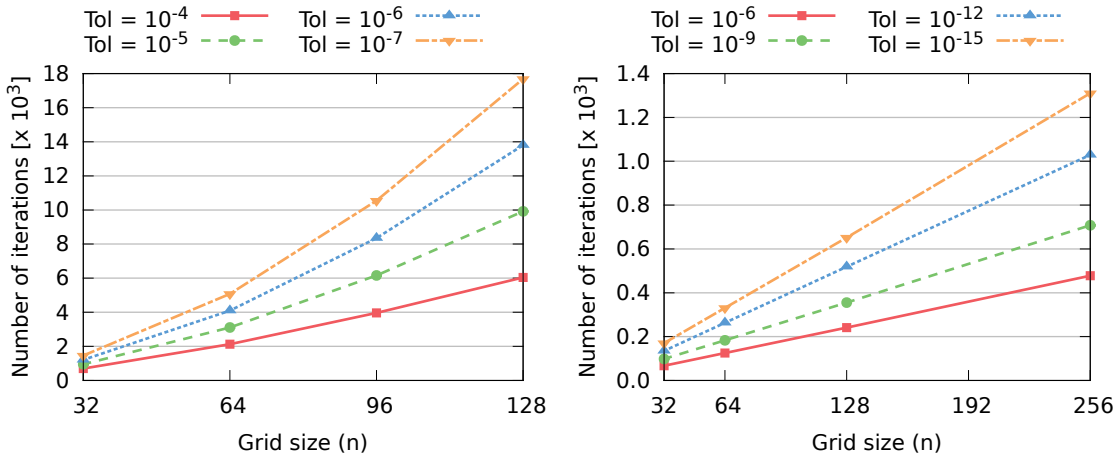
These results hold for the Gauss-Seidel method, however in this case $\rho_{GS} = (\rho_J)^2$, i.e. one Gauss-Seidel iteration decreases the error as much as two Jacobi iterations.

In contrast, when the optimal relaxation parameter is used in the SOR method, the rate of convergence is (Demmel, 1997, §6.5.4)

$$\rho_S = \frac{\cos^2 \frac{\pi}{n+1}}{\left(1 + \sin \frac{\pi}{n+1}\right)^2} \approx 1 - \frac{2\pi}{n+1} = 1 - \mathcal{O}(h). \tag{4.17}$$

In this case, the number of iterations required to decrease the error by a factor of e^{-1} is of order $\mathcal{O}(n)$, and the overall complexity of the SOR solver is $\mathcal{O}(n^3) = \mathcal{O}(N^{1.5})$. This is the same as in the Conjugate Gradients method (Demmel, 1997, p. 277).

The results presented in Figure 4.4 confirm the above-mentioned theoretical results. Indeed, further error reductions require roughly the same number of iterations. In the case of a 128×128 grid and the Gauss-Seidel method, roughly 12,000 iterations are required to reduce the error by a factor of 10^3. In comparison, the SOR method requires only roughly 150 iterations — the consequence of an order of magnitude better convergence rate. As expected, the number of SOR method iterations required to reach a result with specified tolerance grows in linear fashion with increasing n (Figure 4.4B). In contrast, the plots in Figure 4.4A form parabolas confirming an $\mathcal{O}(n^2)$ increase in the case of the Gauss-Seidel method.



(A) Gauss-Seidel method

(B) SOR method with $\omega = \omega_{opt}$

FIGURE 4.4: Iterations required to converge to a solution with the specified error tolerance. Optimal relaxation parameter $\omega_{opt} = 1.9524558$.

The next step was to assess the impact of the Red-Black ordering on convergence of the SOR method. According to Young (1971), this is an example of *consistent ordering* and should not affect the rate of convergence. Indeed, no significant degradation

could be observed regardless of the relaxation parameter value (Figure 4.5). However, the results obtained with different orderings (even consistent) are not identical — in the case of suboptimal ω values, the number of iterations required to reach a specified error tolerance may differ by a constant factor (Figure 4.5 A). In the case of SOR with optimal ω, using the Red-Black ordering may result in a smoother convergence trajectory (Figure 4.5 B).

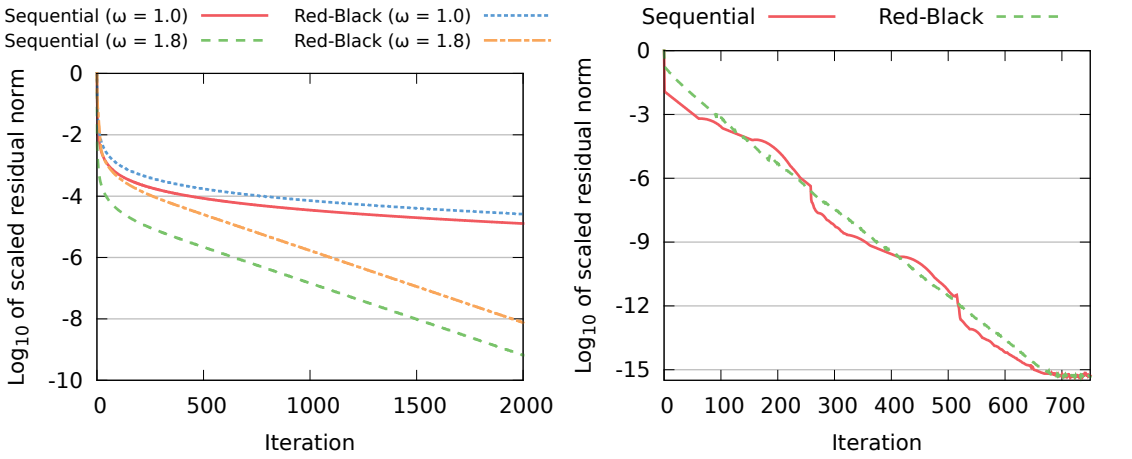
Due to the lack of data dependencies in the Jacobi method, in a distributed memory system each processing element performs the calculation on its assigned subdomain and exchanges halo nodes after each iteration. A similar approach could be used in the parallel SOR implementation, thus avoiding the second exchange in each iteration, however this may lead to unexpected results. When red nodes are updated, the result is exactly the same as in the sequential or shared memory implementation. However, when the black nodes are updated, the red halo nodes are out-of-date. The impact of this phenomenon on SOR convergence is presented in Figure 4.6.

In the case of the Gauss-Seidel method (SOR with $\omega = 1$), the convergence rate is unaffected but as the relaxation parameter increases towards the optimal value, the convergence rate degradation becomes more noticeable, up to the point when the method fails to converge for $\omega = 1.8$ and 16 subdomains (Figure 4.6 A). When the optimal relaxation parameter is used, the parallel SOR method fails to converge even on four subdomains; on two subdomains the convergence rate is significantly degraded (Figure 4.6 B).

To alleviate the adverse effect of out-of-date data, red halo nodes must be exchanged before the black nodes can be updated. This imposes an additional communication overhead, but guarantees that the result is exactly the same as in sequential or shared memory implementations.

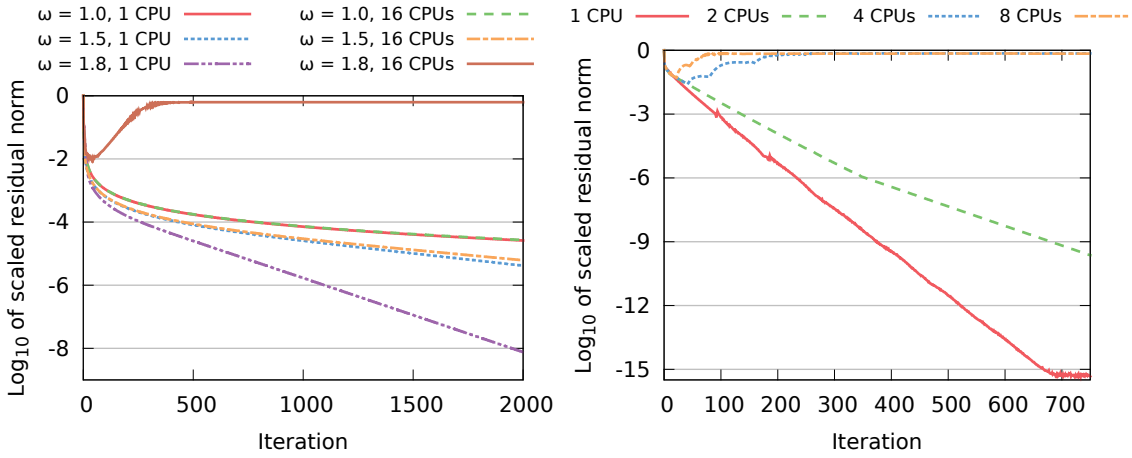
4.3 Implementation Details

This section contains details on implementations of the Successive Over-Relaxation solver for the 2D Poisson's Equation considered in this study: shared memory solvers (Subsection 4.3.1), distributed memory CPU-based solver (Subsection 4.3.2), and distributed memory hybrid CPU-GPU solver (Subsection 4.3.3).



(A) Suboptimal SOR (B) SOR with $\omega = \omega_{opt}$

FIGURE 4.5: Impact of ordering on SOR convergence rate (a 128×128 grid, double precision). The optimal relaxation parameter $\omega_{opt} = 1.9524558$. No significant difference can be observed between the convergence trajectory of sequential and Red-Black orderings.



(A) Suboptimal SOR (B) $\omega = \omega_{opt}$

FIGURE 4.6: Convergence degradation in the MPI SOR implementation with only one halo nodes exchange per iteration (a 128×128 grid, double precision). The optimal relaxation parameter $\omega_{opt} = 1.9524558$. As the relaxation parameter approaches the optimal value, and the number of subdomains increases, the convergence rate deterioration becomes more severe, up to the point when the method fails to converge.

4.3.1 Shared Memory Solvers

The classic SOR formulation (Algorithm 4.1) enables only limited parallelism. To benefit from parallel diagonal processing would significantly complicate the code, therefore only a sequential implementation was considered.

The multi-threaded SOR with the Red-Black ordering (Algorithm 4.2) was implemented with the OpenMP 3.0 library (OpenMP Architecture Review Board, 2008). The grid nodes were equally distributed among the available threads.

On the GPU, the algorithm was implemented in CUDA assigning one thread per grid element. In consequence, each thread has to read six values from global memory and write one value back. While every grid element is read five times, a significant proportion of global memory reads would be excessive if no caching is present. Since the introduction of automatic caching for global memory in devices with Compute Capability 2.0, the use of texture or shared memory for this purpose is unlikely to result in any performance improvement (cf. Subsection 3.2.1). In consequence, all the memory reads were performed directly from global memory. Algorithm 4.4 shows the actual CUDA code for SOR iteration.

Since shared memory is not used, it can be switched to act as an additional L1 cache with `cudaFuncSetCacheConfig` call with the `cudaFuncCachePreferL1` parameter. Indeed, this change resulted in roughly 5% improvement in the overall performance.

In addition to performing SOR iterations, it is necessary to upload the initial grid to the GPU at the beginning, and then to download the result from the GPU at the end. These memory transfers are conducted via the PCI-Express connection and impose an additional communication overhead.

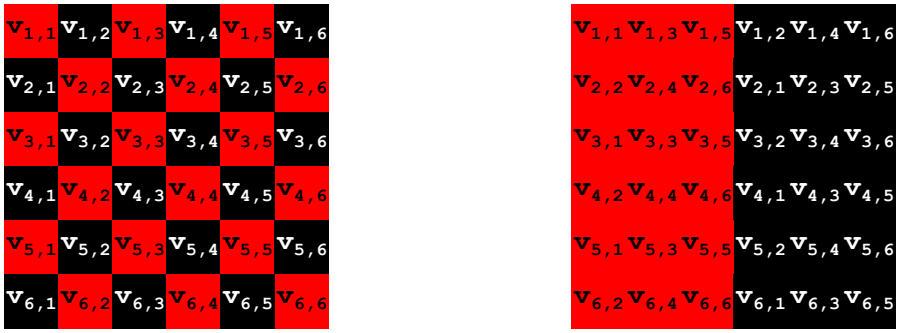
Both parallel implementations were considered with two types of data layout: natural (Figure 4.7A) and separated (Figure 4.7B). The latter layout was motivated by the GPU global memory access performance model — reading every other element in each row may limit the performance due to the less efficient use of coalescing (NVIDIA, 2011, §5.3.2.1). In fact, the separated layout may also improve the performance of CPU implementations due to the more favourable cache access pattern.

Algorithm 4.4 CUDA kernel updating the grid nodes of one colour in the SOR solver

```
template <typename T>
__global__ void Sor(int offset, int width, int height, T h, T w,
                    int f_pitch, const T* f, int g_pitch, T* grid) {
  // Compute which element is updated by this thread
  // Offset depends on the updated colour: 0 for red, 1 for black
  int row = threadIdx.y + blockIdx.y * blockDim.y;
  int col = 2 * (threadIdx.x + blockIdx.x * blockDim.x)
          + ((offset + row) & 1);
  if (row >= height || col >= width) return;

  // Adjust global memory pointers
  f += col + row * f_pitch;
  grid += col + row * g_pitch;

  // Read the adjacent values and update the grid
  T top = (row > 0 ? *(grid - g_pitch) : 0);
  T left = (col > 0 ? *(grid - 1) : 0);
  T right = (col + 1 < width ? *(grid + 1) : 0);
  T bottom = (row + 1 < height ? *(grid + g_pitch) : 0);
  *grid = (T(1) - w) * *grid
        + w/4 * (*f * h * h + top + left + right + bottom);
}
```



(A) Natural layout (B) Separated layout

Figure 4.7: Two approaches to layout of red and black nodes in the memory (stored in the row-major order).

4.3.2 Distributed Memory CPU-based Solver

The distributed memory SOR solver (Algorithm 4.3) was implemented with MPICH 3.0.4 library (`http://www.mpich.org/`, last access: 20 September 2013) to handle the communication between CPUs. In order to reduce the communication overhead, the non-blocking approach was adopted (cf. Subsection 3.1.5). Each iteration begins with the update of top and bottom border nodes. The following calculation on the interior nodes is then overlapped with the exchange of the halo nodes (Figure 4.8).

To minimise the time required for the halo nodes exchange, persistent MPI communication is used. This allows for optimisations leading to better performance (Hatanaka et al., 2013). The pseudo-code for the non-blocking MPI SOR solver is presented in Algorithm 4.5.

Note that the communication overhead scales linearly with increasing P — the number of processing elements (PEs). Each PE sends and receives two vectors of n elements regardless of the P value, i.e. the total size of the data sent over the network by all PEs is $\mathcal{O}(nP)$. Depending on the network topology, it is possible to exchange the halo nodes in parallel, requiring only $\mathcal{O}(n)$ time. Regardless of the number of PEs, the amount of data transferred in the `MPI_Gather` step is of order $\mathcal{O}(n^2)$. This operation cannot be overlapped with any computation, thus increases the overall communication overhead.

4.3.3 Hybrid CPU-GPU Solver

The hybrid SOR implementation is similar to the one described in the previous subsection, however the local grid is stored and processed on the GPU, rather than on the CPU. The main difference is the way the halo node exchange is handled.

All computation on the GPU is performed asynchronously with respect to the CPU. While the interior grid nodes are scheduled for an update on the GPU, the already updated border nodes are downloaded to the CPU memory and then exchanged with the adjacent CPUs. Once the MPI communication is completed, the up-to-date halo nodes are uploaded back to the GPU memory. The CPU-GPU transfers can run concurrently to the interior grid update when calling the asynchronous version of `cudaMemcpy` routine, and separate CUDA streams are used for computation and

Algorithm 4.5 The non-blocking distributed memory parallel SOR (MPI)

Initialise the persistent communication for halo node exchange
{ `MPI_Send_init`, `MPI_Recv_init` }
while Stopping criteria are not met **do**
  { The following four lines are presented in Figure 4.8. }
  Update the border red nodes in the local stripe
  Initialise asynchronous red halo nodes exchange { `MPI_Startall` }
  Update the interior red nodes in the local stripe
  Wait for the halo nodes exchange to complete { `MPI_Waitall` }

  Update the border black nodes in the local stripe
  Initialise asynchronous black halo nodes exchange { `MPI_Startall` }
  Update the interior black nodes in the local stripe
  Wait for the halo nodes exchange to complete { `MPI_Waitall` }
end while
Combine all local stripes to form a final solution { `MPI_Gather` }

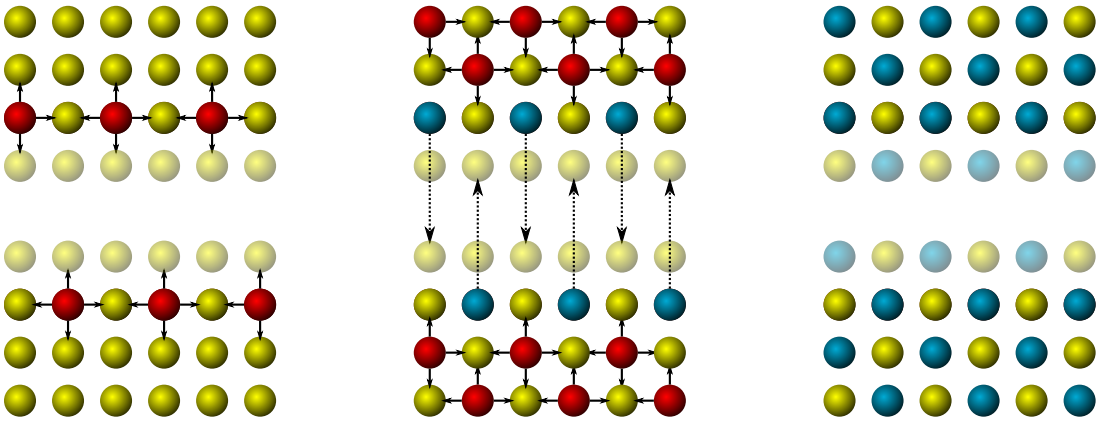


Figure 4.8: The update of red nodes in the distributed memory SOR implementation. The grid nodes processing and halo nodes exchange is overlapped. The halo nodes are transparent, small arrows show data dependencies, dotted arrows show data transfers. Colour legend: yellow — old values, blue — already updated nodes, red — nodes that are updated in parallel.

Algorithm 4.6 The Hybrid CPU-GPU parallel SOR solver

```
Upload the local grid to the GPU
Initialise the persistent communication for halo node exchange
while Stopping criteria are not met do
    Update the border red nodes on the GPU
    Initialise the interior red nodes update on the GPU
    Initialise and wait for the red halo nodes exchange { CPU-GPU and MPI }
    Wait for the GPU to complete the interior nodes update

    Update the border black nodes on the GPU
    Initialise the interior black nodes update on the GPU
    Initialise and wait for the black halo nodes exchange { CPU-GPU and MPI }
    Wait for the GPU to complete the interior nodes update
end while
Download the local stripe from the GPU
Combine all local stripes to form a final solution { MPI_Gather }
```

communication (Harris, 2012). The pseudo-code for the hybrid SOR solver is given in Algorithm 4.6.

Due to the three-stage transfer, the halo node exchange requires more time than in the CPU-based solver. In consequence, it is more difficult to hide this communication overhead — to completely overlap it with computation, the solver would have to be run on larger grids. In addition to the final `MPI_Gather`, transferring the local grid between the CPU and the GPU imposes an additional overhead since these operations cannot be overlapped with any computation.

4.4 Numerical Experiments

All machines used in the experiments were equipped with an Intel Core i7-980x CPU (12 MB cache, 3.33 GHz, six cores), were running the Ubuntu 12.04 Server (64-bit) operating system, and were connected with the Gigabit Ethernet network. The main memory consisted of six 2 GB PC3-8500 DIMMs working in triple channel mode, i.e. the theoretical maximum bandwidth was 25.6 GB/s. The CPU code was compiled with `gcc` version 4.6.3, and the GPU code was compiled with `nvcc` release

5.0, V0.2.1221 and CUDA Toolkit 5.0.35. Technical details of three GPU models (Tesla C1060, Tesla C2050, GTX 480) used in the experiments are summarised in Table B.2 in Appendix B.

Unless specified otherwise, all experiments were run in single and double floating-point arithmetic, were repeated ten times, results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5%. As the experiments were solely focused on performance, the only stopping criterion was the maximum number of iterations, which was set to 1,000.

4.4.1 Performance Models

To update one grid element, the SOR method requires 11 floating-point operations, reads six values, and writes one. Therefore, the computation intensity of the SOR solver is roughly 0.4 FLOPs per byte in single, or 0.2 FLOPs per byte in double precision. In consequence, the problem is memory-bound and the performance is measured in terms of memory throughput. To ensure reliable performance measurement and fair comparison between the algorithms, the following performance model was used for shared memory solvers:

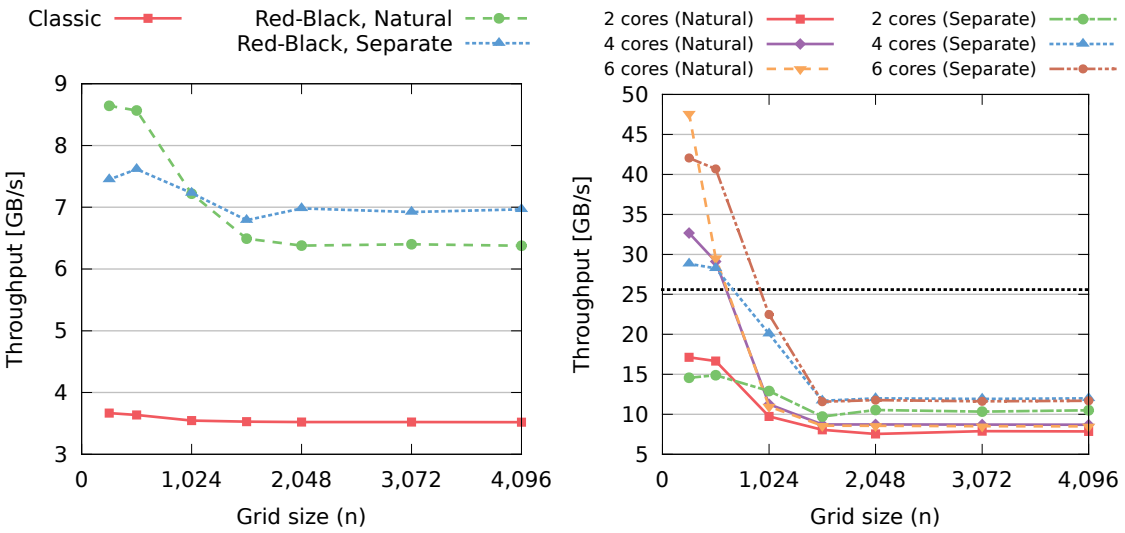
$$F_{SOR}(n, r) = \frac{3Rn^2s}{t_{SOR} \cdot 10^9}. \qquad [GB/s] \tag{4.18}$$

Here n denotes the number of rows and columns in the grid, R is the number of iterations and s is equal to four or eight bytes, depending on arithmetic precision. It is tempting to use a factor of seven rather than three since each grid element update involves seven memory accesses. However, five of these involve shared data. In consequence, each grid element is read five times and the efficient implementation should hide this fact using caching, hence the factor of three — ideally each iteration requires reading the whole grid and source term once, and writing the updated grid.

4.4.2 Multi-threaded CPU Solver

The first experiment compared the classic sequential SOR with two multi-threaded implementations based on the Red-Black ordering and two memory layouts, which are described in Subsection 4.3.1.

The highest memory throughput was observed for small grids ($n \leq 512$). When the grid size increased, the throughput decreased at first and then stabilised (Figure 4.9). This was connected with the CPU cache size — small grids fitted completely in L3 cache greatly reducing the load on the main memory. Once grids got bigger, the nodes had to be read from, and stored to the main memory at each iteration.



(A) Single core (B) Multiple cores

FIGURE 4.9: Performance of the multi-threaded SOR solver on the CPU (double precision). The dotted line marks the maximum main memory bandwidth. For $n \leq 512$, the entire grid fitted in the CPU cache leading to the throughput exceeding that maximum. The performance stabilises once the grids are too large to fit in the L3 cache.

Both implementations based on the Red-Black ordering significantly outperform the classic solver, even when run with only one thread (Figure 4.9 A). This is connected with the L1 cache utilisation. The classic SOR iteration suffers from read-after-write dependency at each node. In the Red-Black ordering, this dependency only occurs when the colour update is finished and the next *half*-iteration begins.

The L1 cache utilisation is also responsible for the relative performance of the Red-Black solver with different memory layouts. To observe the highest performance, at least four grid rows (three grid, and one source term) have to fit in the L1 cache. Since it is 32 KB in size, this requirement is met for $n \leq 1{,}024$ in double precision (4 rows of 1,024 elements, 8 bytes each). Indeed, up to this point, both data layouts

benefit from processing the whole row in L1 cache, but in the case of natural layout the data is closer, i.e. left and right nodes are just next to the updated node, whereas in separated layout they would be roughly $n/2$ elements further on in memory. In consequence, the Red-Black SOR is faster with natural ordering for small grids.

Once n values exceeded 1,024, the SOR solver with separated red and black nodes was faster. The main reason is that in this case, memory accesses involve consecutive addresses, while in the natural layout the data is accessed with the stride of two elements, effectively degrading the cache performance. On a single CPU core, using the separated layout resulted in roughly 10% performance improvement — the observed throughput was up to 7 GB/s.

The superiority of the separated memory layout is even more prominent when multiple threads were used due to the increasing demand for both L1 and L3 caches with the increasing number of threads (Figure 4.9 B). For sufficiently large grids, using the separated layout resulted in roughly 38% performance improvement — the observed throughput up to 12 GB/s, which amounts to 47% of the theoretical maximum bandwidth. It was not necessary to use all six cores to achieve this performance, which confirms that the problem is indeed memory-bound.

The results for the smallest grids ($n = 256$) were different: the solver with the natural layout was faster due to better data locality, similar to the single-core case. Interestingly, throughput exceeding the theoretical maximum bandwidth was observed. This was possible whenever the whole grid and the source term fitted in the L3 cache, i.e. $n \leq 900$ in double precision (12 MB of the L3 cache). In this case, the amount of data transferred from the main memory was greatly reduced, and in consequence, the problem became compute-bound: the observed performance scaled linearly with the number of cores, regardless of the memory layout reaching the throughput of 48 GB/s, or almost 4 GFLOPS in double precision.

Table 4.1 summarises times required to complete 1,000 iterations in single and double precision using both memory layouts. For sufficiently large grids, using double precision resulted in doubled execution time since the amount of processed data increased two-fold. For the model problem considered in this study, the number of iterations required by the optimal SOR to converge in single precision ($res < 10^{-6}$) is roughly $2n$, and in double precision ($res < 10^{-15}$) roughly $5n$ (Figure 4.4 B). In

Table 4.1: Multi-threaded SOR solver execution times (in milliseconds). The solver was run on six cores of Intel i7-980x for 1,000 iterations. A significant performance drop can be observed when grid is too large to fit in L3 cache ($n = 1{,}024$ for single, and $n = 512$ for double precision). Beyond that threshold, changing from single to double precision doubles the execution time since the problem is memory-bound. For sufficiently large grids, the separation of red and black nodes leads to higher cache hit ratio, thus faster execution.

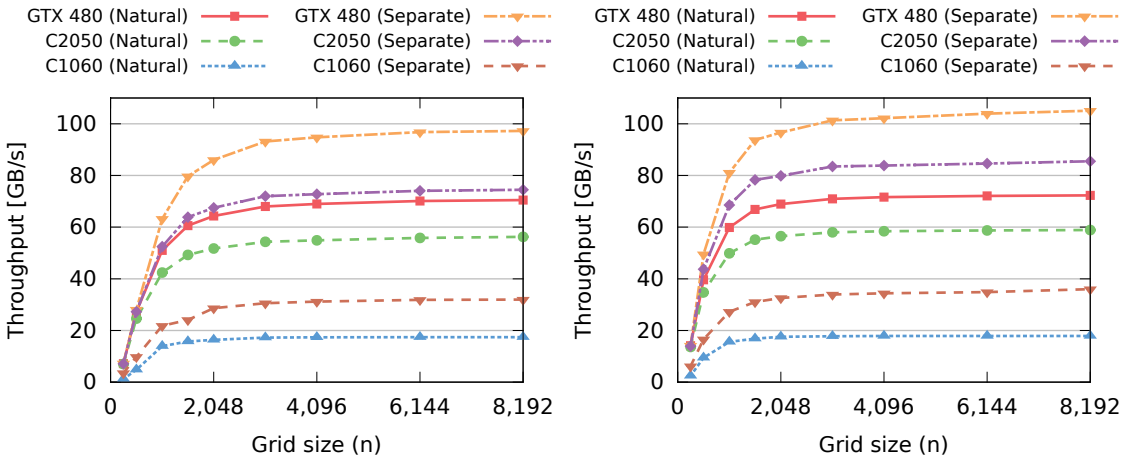
Grid size (n)	Single precision		Double precision	
	Natural	Separated	Natural	Separated
256	36.7	40.8	35.6	40.3
512	141	147	221	161
1,024	571	582	2,337	1,141
1,536	2,866	2,011	6,690	4,955
2,048	5,942	4,311	11,879	8,637
3,072	13,379	9,976	26,754	19,610
4,096	23,781	17,328	47,625	34,609

consequence, for the largest grid considered in this experiment, consisting of over 16 million variables, the multi-threaded SOR solver requires roughly 140 seconds to converge in single precision, and roughly 700 seconds in double precision.

4.4.3 Single GPU Solver

The next experiment aimed at investigating the impact of memory layout on the SOR solver performance on the GPU. Similar to the CPU implementation, for sufficiently large grids the observed throughput did not differ significantly between the single and double precision (Figure 4.10). The peak performance was achieved only when all GPU cores were fully utilised.

For small grids, the observed throughput in double precision (Figure 4.10 B) is significantly higher than in single precision (Figure 4.10 A), e.g. 81 GB/s vs. 63 GB/s for a $1{,}024 \times 1{,}024$ grid on GTX 480. While the number of running threads is the same



(A) Single precision (B) Double precision

FIGURE 4.10: Performance of SOR iterations on a single GPU. The solver achieves up to 60% of the theoretical peak performance on devices with global memory caching (GTX 480, Tesla C2050) and up to 35% on the older device (Tesla C1060). Separating red and black nodes results in significantly better performance — by 38% on more recent, and up to 100% on older GPUs, due to the reduced amount of global memory read in excess.

— and too low to fully utilise the GPU — twice as much data is transferred in the case of the double precision solver resulting in the higher global memory saturation.

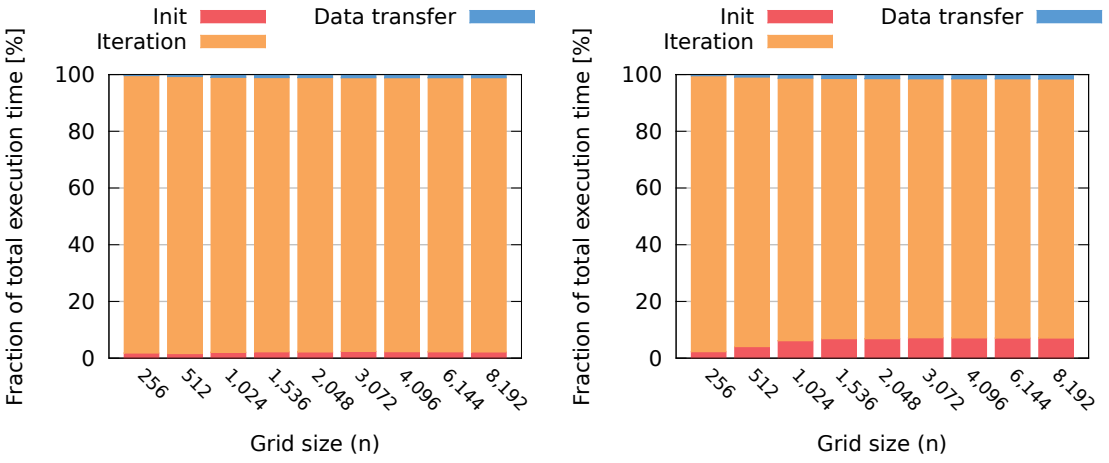
Separating red and black nodes resulted in a significant performance improvement due to better global memory bandwidth utilisation. On Tesla C1060 (no global memory caching), the improvement is almost two-fold due to the more efficient pattern of global memory reads. Accessing a single value from global memory can be expensive as the latency is typically 300–600 GPU cycles. To alleviate this limitation, global memory is accessed in transactions of 32-, 64-, or 128-bytes, which allow us to *coalesce* reading multiple global memory values as long as they are accessed by 32 threads in the same warp. When consecutive values are accessed, only one (two in double precision) 128-bytes transaction is required and no excess data is read. However, this is not the case with the natural data layout — the data is accessed with a stride of two values resulting in 50% data read in excess, effectively limiting the observed throughput (cf. Appendix A and NVIDIA, 2011, §F.4.2).

Interestingly, the performance improvement on devices with Compute Capability 2.0 is smaller, and amounts to roughly 38%, which is the same as in the CPU

implementation. In this case, the performance penalty from reading the same values multiple times is partially hidden due to global memory caching. However, the penalty from reading excess data in the case of the natural data layout remains and causes the difference in the observed memory throughput.

The GPU implementation of the SOR solver achieved memory throughputs of up to 105 GB/s on GTX 480. This amounts to roughly 60% of the theoretical maximum bandwidth (177 GB/s). The same efficiency was observed on Tesla C2050 (85.5 out of 144 GB/s). In contrast, on the older GPU with Compute Capability 1.3, the achieved throughput amounted to only 35% (36 out of 102 GB/s). While devices with Compute Capability 2.0 are equipped with L1 and L2 caches large enough to fit multiple grid rows, the Tesla C1060 has no global memory caching. The impact of caching on performance was investigated in detail in Subsection 3.2.1. The results indicate, that if texture memory was used for caching, then the throughput on Tesla C1060 is likely to reach 60% of the theoretical maximum, i.e. roughly 61 GB/s.

Performing iterations was not the only cost of the GPU implementation. The GPU memory had to be allocated, the initial grid had to be uploaded to the GPU, and the final result had to be downloaded back to the CPU.



(A) Natural layout

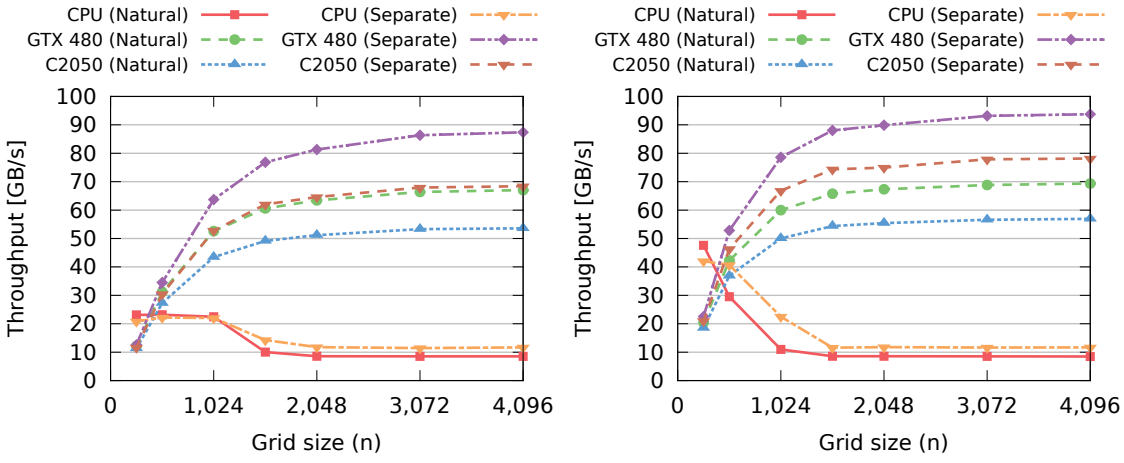
(B) Separated layout

Figure 4.11: Execution time breakdown in the SOR solver on a single GPU (GTX 480, double precision). The initialisation and communication phases have relatively small impact on the overall performance. The separation and merge of red and black nodes in (B) is responsible for a higher fraction of time spent on initialisation.

The initialisation and communication phases had a small impact on the execution time (Figure 4.11). The CPU-GPU communication — performed through PCI-Express at 6–6.6 GB/s — was responsible for 1–2% of the total execution time.

In the SOR solver with natural memory layout, the initialisation consisted only of GPU memory allocation and amounted to 2–2.5% of the total execution time (Figure 4.11A). In contrast, when separated layout was used, the data had to be transformed before uploading to the GPU and then merged again at the end. This imposed an additional 5% overhead.

Low device utilisation and additional overheads in the GPU implementation are responsible for low performance observed on small grids (Figure 4.12). The GPU solver was significantly faster than the CPU implementation only for sufficiently large grids ($n \geq 1{,}024$). In this case, the overheads were negligible and the overall performance was similar to throughput observed during the iteration phase.



(A) Single precision (B) Double precision

FIGURE 4.12: Comparison of the SOR solver performance on an Intel i7-980x CPU and two GPUs. CPU-GPU data transfer time was taken into account. For sufficiently large grids, the GPU implementation on GTX 480 is up to eight times faster than SOR on the CPU. When $n \leq 512$, the GPU solver is slower due to relatively high overheads.

Table 4.2 contains execution times of the SOR solver on the fastest GPU (GTX 480) and compares them with results observed for the multi-threaded CPU implementation. The differences in throughput presented in Figure 4.12 had a direct reflection

Table 4.2: SOR solver execution times on a single GPU. The solver was run with separated nodes on a GTX 480 GPU for 1,000 iterations. CPU-GPU data transfer time was taken into account. Speed-ups were measured against the fastest CPU implementation running on six cores of an Intel i7-980x CPU.

	Single precision		Double precision	
Grid size (n)	Time [ms]	Speed-up	Time [ms]	Speed-up
256	64	0.60×	71	0.53×
512	92	1.56×	120	1.30×
1,024	198	2.89×	321	3.49×
1,536	369	5.39×	644	7.61×
2,048	620	6.90×	1,122	7.64×
3,072	1,313	7.56×	2,433	8.02×
4,096	2,305	7.49×	4,300	8.02×
6,144	5,097	—	9,525	—
8,192	9,014	—	16,756	—

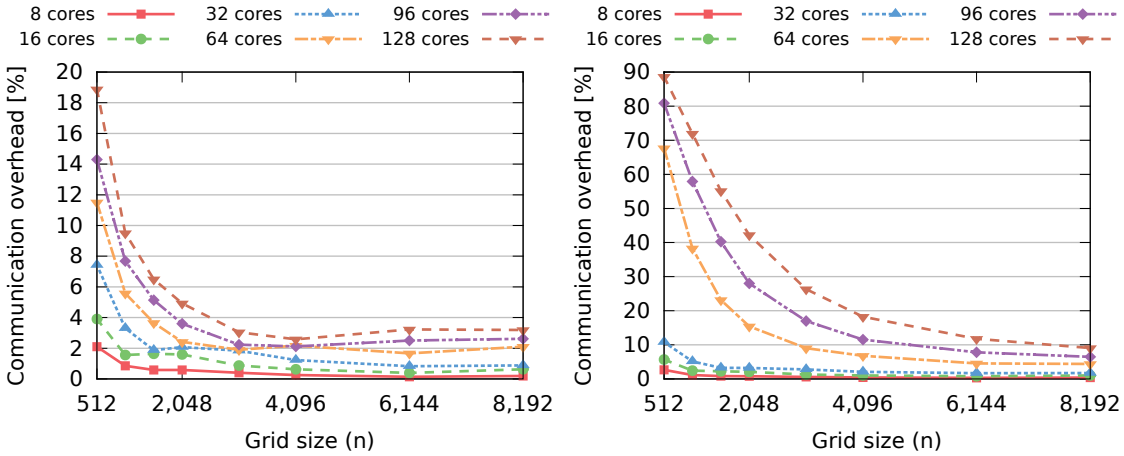
in the execution times: GPU solver is up to 7.5 times faster than the CPU solver in single precision, and up to 8 times in double precision. In comparison, the theoretical maximum bandwidths of both platforms differ by a factor of 6.93. This means that the GPU implementation utilised the available resources better.

4.4.4 Distributed CPU Solver on Large Cluster with Infiniband

To assess the performance and scalability of the distributed memory parallel SOR solver, the experiments were conducted on the Cranfield University supercomputer — Astral. It is equipped with 80 computing nodes, each one consisting of two Intel Xeon E5-2660 CPUs (20 MB cache, 2.20 GHz, 8 cores) and 64 GB of main memory, i.e. each node consisted of 16 cores and 4 GB of memory per core. The memory on each node was organised in eight 8 GB PC3-12800 DIMMs working in

dual channel mode — the theoretical maximum bandwidth was 25.6 GB/s. The nodes were connected with Infiniband low-latency network (up to 40 Gbit/s).

As explained in Subsection 4.3.2, in a distributed memory environment additional communication is required: the halo node exchange twice per iteration, and gathering the final result at the end (`MPI_Gather`). The first step in experiments was to assess the impact of the communication overhead on the overall performance. The results are presented in Figure 4.13.



(A) Halo nodes exchange

(B) Total communication overhead

Figure 4.13: Communication overhead in the Distributed SOR Solver on Astral (separated layout, double precision). For sufficiently large grids ($n \geq 3{,}072$), the halo nodes exchange can be completely overlapped with computation. The remaining overhead in (A) comes from MPI processes synchronisation. Gathering the final result has a significant impact on the overall performance, which decreases with the increasing grid size. Further decrease could be observed if the number of iterations was increased.

To minimise the communication overhead, the SOR implementation was designed to overlap the halo node exchange with an update of interior nodes. This was not possible when processing small grids, but once n values exceeded 2,048, the overhead dropped below the 5% level, but did not converge to zero (Figure 4.13 A).

For sufficiently large grids the communication overhead is no longer connected with data transfers, which are completely overlapped with computation, but rather with the cost of MPI operations: issuing the asynchronous send and receive, and then syn-

chronisation at the end of each exchange. This explains why the overhead increased with the number of cores — the cost of synchronisation increased as well.

Similar trends could be also observed when the total communication overhead was considered — halo nodes exchange and gathering the final result. However, this time the fractions of the total execution time were significantly higher (Figure 4.13 B). This is caused by the latter operation, which could not be overlapped with any computation. It was performed only once after all the iterations are completed, thus its impact on the overall performance depends mostly on the number of iterations rather than the grid size. Moreover, the relative overhead increased with the number of processing elements, however this time significant differences could be observed when 64 cores or more were used. This was connected with the fact that the computing nodes of Astral are connected to four separate network switches — once four or more nodes were used, some of them were connected to a different switch, effectively decreasing the observed throughput by a factor of four.

Communication overheads had a direct impact on performance of the Distributed SOR Solver (Figure 4.14). When two or three Astral nodes were used, the speed-ups were close to linear, regardless of the grid size (Figure 4.14 A). For small grids ($n \leq 2{,}048$), the parallel efficiency decreased steadily when further nodes were added due to high communication overhead caused by the `MPI_Gather` operation. For large grids, close to linear speed-ups were observed even when 128 cores were used.

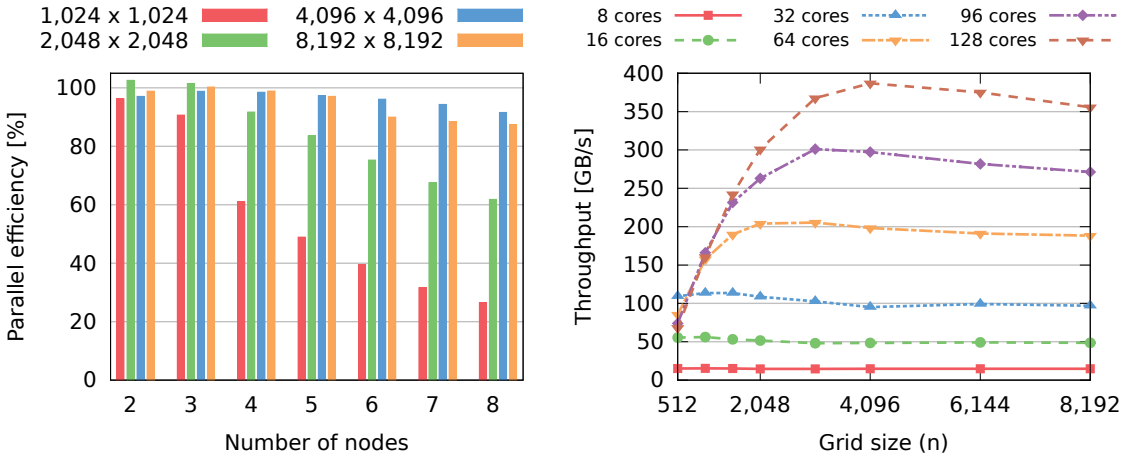
For a fixed size grid, increasing the number of processing elements had two consequences: an increase in the fraction of time spent on communication, and a possible performance gain due to smaller local data chunks that can better fit into the CPU cache. This explains why the parallel efficiency stayed at a high level for large grids.

The local data (the grid and the source term) processed by each CPU was stored in two $n \times \frac{n}{2P} \times s$ bytes arrays, where P is the number of Astral nodes (2 CPUs each), and s is 4 or 8 bytes depending on arithmetic precision. If C denotes the CPU L3 cache size (in bytes), the local chunks fitted in the cache completely wherever

$$2\,n\,\frac{n}{2P}\,s \leq C, \tag{4.19}$$

which can be solved for n, giving

$$n \leq \sqrt{\frac{PC}{s}}. \tag{4.20}$$



(A) Parallel efficiency (B) Processing speed

FIGURE 4.14: Performance of the Distributed SOR Solver on Astral (separated layout, double precision). Astral nodes comprise 16 CPU cores. Communication overheads increased with the number of cores, but ware partially compensated by higher CPU cache hit ratio. For sufficiently large grids, the speed-ups are close to linear even on 128 cores.

As was shown in Subsection 4.4.2, whenever the local grid and source term chunks fit in L3 cache, the SOR iterations can be performed much faster and may achieve memory throughput higher than the theoretical maximum main memory bandwidth. The CPUs installed in Astral had 20 MB of L3 cache, therefore even for relatively large grids local grid and source term chunks could fit in completely.

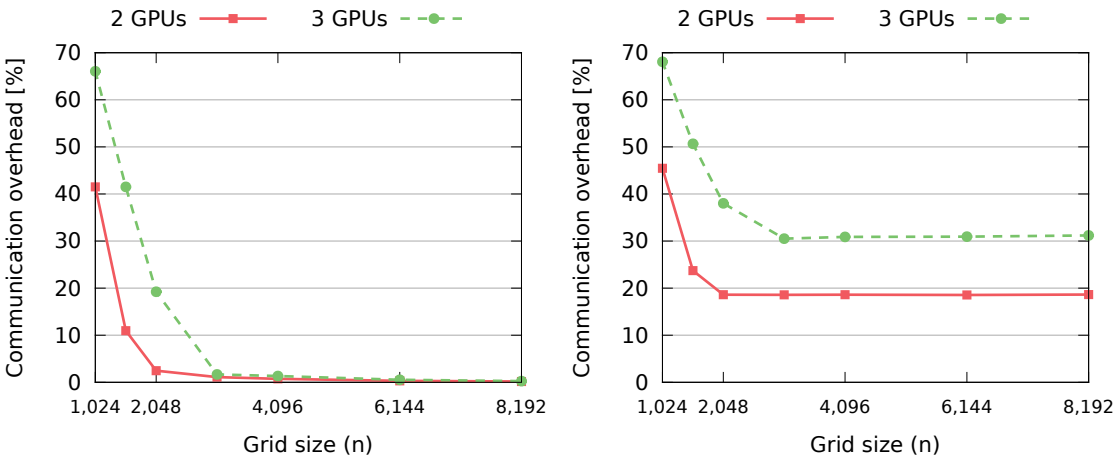
When eight Astral nodes were used, in double precision grids up to $n = 4{,}096$ could fit in the cache. This explains why the parallel efficiency was higher for $n = 4{,}096$, in comparison to $n = 8{,}192$, when six or more nodes were used for computation. The local chunks of the former grid fitted in the cache, while in the latter case, some values had to be accessed in the main memory in each iteration.

The observed throughput was at a constant level regardless of the grid size when up to 32 cores were used due to relatively small communication overheads (Figure 4.14B). When more computing nodes were used, the peak performance was only observed for sufficiently large grids. In accordance with Equation (4.20), on 8 Astral nodes (128 cores) the best performance was observed for grids consisting of $4{,}096 \times 4{,}096$ elements. On larger grids, the overall performance started to decrease due to the lower cache hit ratio and higher communication overheads.

Interestingly, when solving a grid with $n = 4{,}096$ on eight Astral nodes, the observed throughput was at roughly 48 GB/s per node, which corresponds to the performance observed in Subsection 4.4.2 when all the data fitted into CPU cache (the maximum main memory bandwidth was the same in both cases). In consequence, the whole solver was capable of processing data at 387 GB/s. The performance decreased to 356 GB/s when larger grids ($n = 8{,}192$) were considered due to the above-mentioned caching effects and increasing communication overheads.

4.4.5 Hybrid Solver on Small GPU Cluster

The final set of experiments was conducted to assess the performance of the Hybrid SOR Solver, that combines the distributed memory approach with the GPU processing. As in the previous subsection, the first step was to assess the communication overheads. The results are presented in Figure 4.15.



(A) Halo nodes exchange

(B) Total communication overhead

FIGURE 4.15: Communication overheads in the Hybrid SOR Solver (separated layout, double precision). For sufficiently large grids ($n \geq 3{,}072$) it was possible to completely overlap the halo nodes exchange with computation. The cost of the final `MPI_Gather` operation is proportional to the grid size and had a significant impact on the total execution time, due to relatively slow Gigabit Ethernet interconnect.

Similar to the Astral implementation, the halo node exchange was overlapped with computation. In this case, more time was required for the exchange. In addition

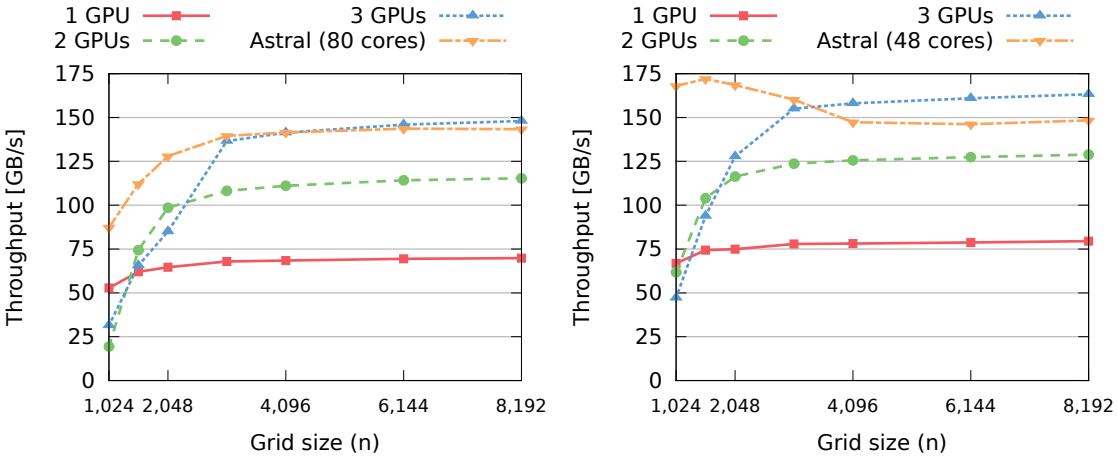
to MPI send/receive operations, the old halo nodes had to be downloaded from the GPU and then the new ones had to be uploaded back. However, the CPU-GPU communication is relatively fast (cf. Subsection 4.4.3).

The halo node exchange overhead was only significant when small grids were considered due to relatively fast grid node updates. Once n values were at least 3,072, the overhead became negligible (Figure 4.15 A). In contrast, the cost of the final `MPI_Gather` operation was relatively high even for the largest grids (Figure 4.15 B). The time required to gather the final solution is proportional to the grid size (n^2). Since the number of iterations was fixed, the fraction of the total execution time spent on `MPI_Gather` was constant. It would only decrease when a larger number of iterations was performed. As was shown in Subsection 4.4.3, the GPU processing is roughly 8 times faster than on the CPU, therefore the iterations are relatively quick and the time spent on MPI communication is significant — 20% of the total execution time on two GPUs, and 30% on three graphics cards.

Due to the above-mentioned impact of `MPI_Gather` operation on the overall throughput, when the solver was run for only 1,000 iterations, the performance of the Hybrid SOR Solver was measured both with and without time required to gather the final result (Figure 4.16). The latter approach gives an unbiased picture of the prospective performance on bigger GPU clusters with faster interconnect. Due to the halo node exchange overhead, the maximum performance was observed only for $n \geq 3{,}072$.

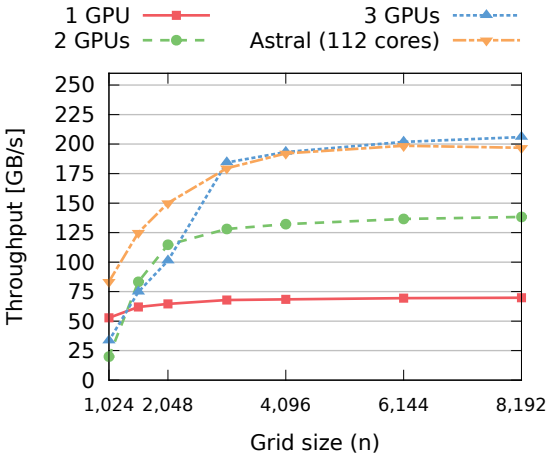
When the cost of `MPI_Gather` was included, the speed-up with respect to the number of GPUs was far from linear, however a significant increase in overall performance could still be observed. In single precision, the hybrid solver on 3 GPUs processed data at up to 150 GB/s, compared to roughly 70 GB/s on a single GPU (Figure 4.16 A). The former result is comparable to the performance observed on 80 cores of Astral. In double precision, the performance was slightly better — over 75 GB/s on a single C2050, and 165 GB/s on three GPUs (Figure 4.16 B). Due to more significant differences between single and double precision on Astral, the performance on three GPUs could be matched by only 48 Astral cores.

In order to get a clearer picture of the hybrid SOR performance, and provide fairer comparison with Astral performance, the time spent on `MPI_Gather` was omitted. Indeed, Astral benefits from fast Infiniband interconnect, while the hybrid solver was run on a significantly slower Gigabit Ethernet network.

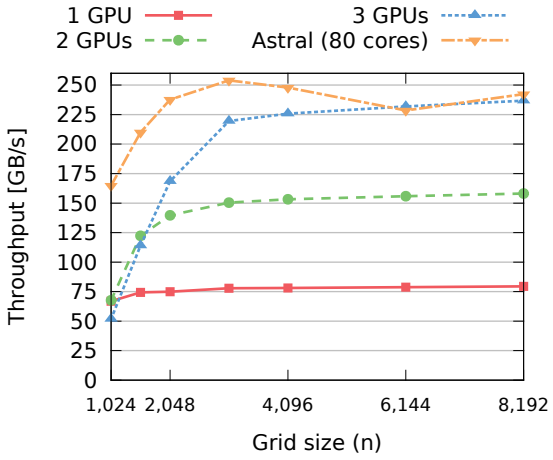


(A) Single precision (`MPI_Gather` included)

(B) Double precision (`MPI_Gather` included)



(C) Single precision (`MPI_Gather` excluded)



(D) Double precision (`MPI_Gather` excluded)

Figure 4.16: Performance of the Hybrid SOR Solver (Tesla C2050, separated layout).

Gathering the final result was the main scalability-limiting factor. The linear speed-up was observed, regardless of the precision used. In consequence, the Hybrid SOR Solver on three Tesla C2050 GPUs delivered 207 GB/s in single precision (Figure 4.16c) and matched the performance of 112 cores of Astral. In double precision, the throughput observed on the small GPU cluster was 237 GB/s and matched the performance of 80 cores on Astral.

4.4.6 Comparison with the Existing GPU Implementation

Recently, Di et al. (2012) presented a novel parallel SOR method, namely Multi-layer SOR (MLSOR). The authors claim that their approach admits more efficient data-parallel execution on the GPU than SOR with the Red-Black ordering (RBSOR). Di et al. parallelised the K-layer SOR method by applying a non-dependence-preserving scheme: a new domain decomposition, and the alternate tiling technique.

To verify the performance of their approach, Di et al. compared their results with the Red-Black SOR implementation following Java Grande benchmark suite (Yan et al., 2009). The experiments were conducted on a $8{,}192 \times 8{,}192$ grid, using a Tesla C1060 device with Compute Capability 1.3. In each test run, only 78 iterations were performed, reducing the residual norm by a factor of 10^{-3}.

In order to compare our CUDA implementations, corresponding experiments were conducted with both natural (NATSOR) and separated (SEPSOR) layouts. The results are presented in Figure 4.17. Di et al. did not specify whether they included initialisation and CPU-GPU communication in their time measurement, therefore time spent on computation only, and the total execution time are provided.

MLSOR was faster than NATSOR and comparable to SEPSOR, assuming that Di et al. had taken the initialisation and communication overheads into account. Furthermore, the authors solve Laplace's, rather than Poisson's Equation (they explicitly assume $f \equiv 0$), which is in fact easier — the amount of data read from global memory is reduced by roughly 14%, and the number of FLOPs per grid node decrease from 11 to 8. Moreover, our solver was optimised for devices with Compute Capability 2.0 and higher, thus no explicit global memory caching was implemented. This is why our NATSOR is in fact slower than the equivalent RBSOR. Adding texture memory caching would increase the throughput by roughly 70%, effectively

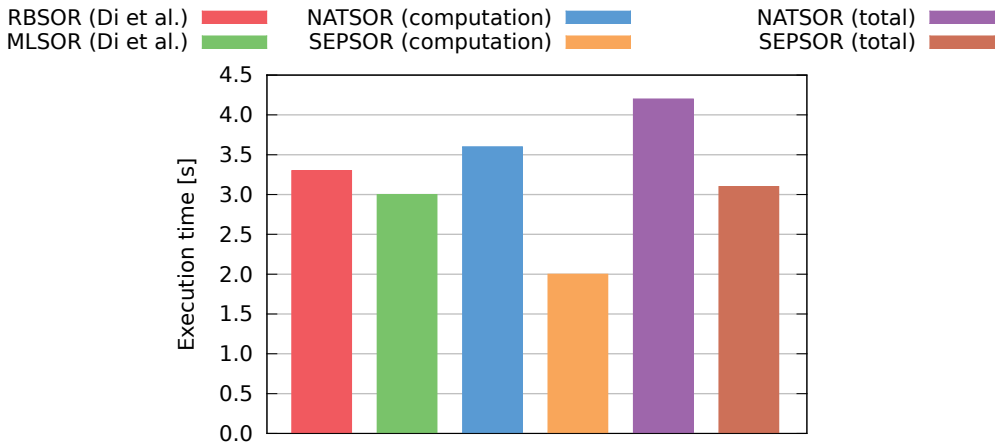


Figure 4.17: Execution times comparison with Multi-layer SOR by Di et al. (2012). The experiment was run on Tesla C1060 on an 8,192×8,192 grid in single precision. Solvers were run for 78 iterations. NATSOR and SEPSOR are not optimised for older GPUs, and suffer from the lack of explicit global memory caching. However, the SEPSOR execution time (including all initialisation and communication overheads) matches that of MLSOR.

reducing the time required for computation by 40%. Finally, in the experiment only 78 SOR iterations were performed — the cost of initialisation and communication was relatively high in comparison to time spent on computation. This was especially prominent in the SEPSOR solver, as separation and merge of grid nodes took almost 600 ms, i.e. 19% of the total execution time. In consequence, our approach — separating red and black nodes in SOR — is likely to provide better overall performance than a significantly more complex multi-layer scheme proposed by Di et al. (2012).

4.5 Conclusions

- This chapter offers several implementations of the parallel SOR solver for 2D Poisson's Equation. SOR is an iterative method capable of matching the convergence rate of Conjugate Gradients provided the optimal relaxation parameter is used. In the general case, the optimal value is not known but it can be explicitly computed when solving linear systems coming from the FDM discretisation of Poisson's Equation. The detailed analyses of the SOR method convergence rate and time complexity are provided.

- The sequential SOR algorithm has to be modified in order to enable parallelism since the classic formulation is inherently sequential. Data dependencies can be decoupled by using the Red-Black ordering. Since that ordering is *consistent*, the convergence rate is asymptotically the same as with the classic ordering.

- Two data layouts were considered in this study — natural, and a novel approach based on the separation of red and black nodes. The latter layout aims at improving the CPU cache utilisation, and reducing the amount of excess data loaded during coalesced reads from global memory on the GPU.

- The significant impact of the new layout was observed in the multi-threaded SOR implementation run on Intel i7-980x. The memory throughput of 12 GB/s (47% of the theoretical maximum) was observed when using the separated layout — 38% higher than in the case of the natural layout. To solve the model problem on a $4{,}096 \times 4{,}096$ grid to the machine error tolerance, the multi-threaded implementation required 140 seconds in single precision, and 700 seconds in double precision.

- SOR with the natural layout was only faster for grids that could completely fit in the L3 cache due to better data locality. In this case, the demand on main memory bandwidth was significantly lower and the problem became compute-bound — indeed, the memory throughput scaled in linear fashion with the number of cores up to 48 GB/s.

- The GPU implementation of the SOR solver was optimised for devices with Compute Capability 2.0 and higher. Similar to the CPU case, the solver using separated layout was 38% faster. The observed throughput was up to 105 GB/s on GTX 480, and up to 86 GB/s on Tesla C2050. On both GPUs, this amounts to 60% of the theoretical maximum global memory bandwidth.

- On an older GPU with Compute Capability 1.3 (Tesla C1060), the proposed implementation achieved only 35% of the theoretical maximum (36 GB/s) due to the lack of automatic global memory caching. In this case, the implementation with separated red and black nodes is two times faster than when the natural layout is used instead. The performance of the SOR solver on older devices can be improved by using texture memory.

- Even taking the CPU-GPU communication overhead into account, the GPU implementation was up to 8 times faster on a GTX 480 GPU, than on an Intel i7-980x CPU. In consequence, the above-mentioned model problem could be solved in 18 seconds in single precision, and 88 seconds in double precision.
- In comparison to the recent SOR implementation replacing the Red-Black ordering with the multi-layer scheme (Di et al., 2012), our implementation with separated red and black nodes provides better performance due to more efficient GPU global memory access. It is also much easier to implement.
- The scalability and performance of the distributed memory implementation of the SOR solver was tested using 128 cores of the Astral supercomputer. Due to the large 20 MB L3 cache on CPUs installed in Astral, the local data chunks of large grids (up to $n \leq 4{,}096$) could fit completely. In consequence, the throughput of 48 GB/s per node was observed. Once the grids are too large to fit in L3 cache, the throughput was reduced to 44 GB/s ($n = 8{,}192$).
- Running the Distributed SOR Solver imposed additional communication overheads. However, for sufficiently large problems close to linear speed-ups were observed, indicating good scalability of the proposed method. The SOR solver was running at up to 387 GB/s on 128 Astral cores. The model problem could be solved in 7 seconds in single precision, and 20 seconds in double precision.
- The Hybrid SOR Solver combines the distributed memory communication used in the Astral implementation with grid processing on the GPU. Even though in this case the halo nodes exchange requires more time, it is possible to completely overlap it with computation for sufficiently large grids ($n \geq 3{,}072$).
- No overlapping is possible in the case of the final `MPI_Gather` operation. Since the machines forming the GPU cluster were connected with the relatively slow Gigabit Ethernet network, this step took a significant fraction of the total execution time due to the fixed number of iterations in experiments. When the cost of gathering the final result was not taken into account, the linear speed-up with respect to the number of GPUs was observed. On just three Tesla C2050 GPUs, the Hybrid SOR Solver was capable of achieving 207 GB/s throughput in single precision, and 237 GB/s in double precision. The model problem could be solved in 8, and 34 seconds, respectively.

Chapter 5

Case Study: Multistart Tabu Search for Quadratic Assignment Problem on CUDA Platform

Relatively little research has been done so far on GPU implementations of algorithms for computationally demanding discrete optimisation problems. In this chapter, the well-known $\mathcal{NP}$-hard Quadratic Assignment Problem (QAP) is considered. Detailed analysis of parallelisation possibilities, and memory organisation and access patterns, allowed us to design a novel, fast meta-heuristic for the QAP problem, intended for GPU platforms — Parallel Multistart Tabu Search (PMTS). Computational experiments showed that PMTS is capable of providing good quality (often optimal or the best known) solutions in a short time.

The initial findings have been presented at the 4^{th} International Many-core and Reconfigurable Supercomputing Conference, MRSC 2011 in Bristol, UK. After the positive feedback, results have been extended and published in Journal of Parallel and Distributed Computing (Czapiński, 2013).

5.1 Introduction

The Quadratic Assignment Problem (QAP) was first defined by Koopmans and Beckmann (1957). In QAP n facilities have to be assigned to n locations in order to minimise the total *flow* (people migration, data transfers etc.) between the facilities.

Distances between the locations are stored in an $n \times n$ matrix d and flows between facilities in an $n \times n$ matrix f. The total cost of any assignment (permutation of facilities) π can be calculated as:

$$C(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{i,j} \cdot f_{\pi(i),\pi(j)}. \tag{5.1}$$

The objective is to find an assignment π^* that minimises the cost function.

The most popular application for QAP is facility layout, e.g. planning university campus (Dickey and Hopkins, 1972), hospital facilities (Elshafei, 1977; Krarup and Pruzan, 1978), or more recently, DNA micro-array layout (de Cravalho Jr. and Rahman, 2006). Moreover, QAP was used to solve the Steinberg wiring problem (Steinberg, 1961; Brixius and Anstreicher, 2001), to design a typewriter keyboard (Pollatschek et al., 1976), and even for dartboard design (Eiselt and Laporte, 1991). A recent survey on QAP history and applications is provided by Loiola et al. (2007).

QAP belongs to the class of $\mathcal{NP}$-hard problems (Garey and Johnson, 1979). In the QAP case it is feasible to compute the exact solution only for relatively small instances with n values up to 30 (Loiola et al., 2007). This limitation raised interest in meta-heuristics for finding near-optimum solutions for QAP. These include algorithms based on Tabu Search (Taillard, 1991), Simulated Annealing (Connolly, 1990), Scatter Search (Cung et al., 1997), Genetic Algorithms (Drezner, 2003), Memetic Algorithms (Merz and Freisleben, 2000), Ant Colony Optimisation (Gambardella et al., 1999; Stützle and Dorigo, 1999) or Iterated Local Search (Stützle, 2006). Even with such sophisticated techniques, the time required to obtain good quality solutions varies from several minutes for medium size datasets ($n \sim 50$) to a few hours for the large ones ($n \sim 100$; James et al., 2009).

The introduction of CUDA and very powerful GPU hardware designed for high-performance computing, have created an opportunity to considerably speed up computation in the above-mentioned algorithms. Several papers have been published on GPU techniques for meta-heuristics for discrete optimisation problems. Janiak et al. (2008) proposed Tabu Search for Travelling Salesman and Permutation Flowshop Problems, but using a pre-CUDA API (Microsoft XNA). Luong et al. (2009) proposed a GPU implementation of Tabu Search for QAP. The shortcoming of both implementations is that only the neighbourhood evaluation is done on the GPU. The

remaining operations are performed on the CPU, effectively limiting the performance by introducing CPU-GPU data transfers in each iteration. The solution considered in our previous study on CUDA Tabu Search for the Permutation Flowshop Problem (Czapiński and Barnes, 2011) alleviates that problem by introducing a GPU implementation of the tabu list. Finally, the CUDA implementation of Tabu Search for QAP (SIMD-TS) proposed by Zhu et al. (2010), employs non-cooperative parallelism by running multiple independent Tabu Search instances. Since Tabu Search is deterministic, certain randomness is introduced to diversify the search paths.

This chapter introduces the Parallel Multistart Tabu Search for QAP, which aims to combine the effectiveness of the Multistart Tabu Search approach with the high performance provided by GPU hardware. Unlike SIMD-TS proposed by Zhu et al. (2010), our algorithm is deterministic and it benefits from the information exchange between parallel Tabu Search instances. Section 5.2 presents the development of the Parallel Multistart Tabu Search method. The GPU implementation of PMTS is described in Section 5.3, then in Section 5.4 results of computational experiments are presented and discussed. Finally, the conclusions are given in Section 5.5.

5.2 Parallel Multistart Tabu Search for QAP

In this section, the classic Tabu Search meta-heuristic is presented, along with motivation behind porting it on the GPU (Subsection 5.2.1). Then, in Subsection 5.2.2, the Multistart Tabu Search and Parallel Multistart Tabu Search schemes are described. Finally, Subsection 5.2.3 presents an algorithm based on the PMTS scheme that finds solutions to the Quadratic Assignment Problem.

5.2.1 Classic Tabu Search

Various modifications of the Tabu Search method proved to be among the best performing meta-heuristics when applied to QAP (Taillard, 1991; Battiti and Tecchiolli, 1994; James et al., 2009). Furthermore, unlike some other meta-heuristics (e.g. Simulated Annealing), Tabu Search involves independent evaluation of several solutions at the same time, which enables parallelisation.

Tabu Search was first proposed by Glover (1986). It is a neighbourhood-based, iterative improvement meta-heuristic that can be applied to many optimisation problems. In every iteration of Tabu Search, an attempt is made to improve the *current solution* by searching its *neighbourhood* and choosing the solution with the best value. To avoid getting stuck in local minima a mechanism called the *tabu list* is introduced.

The performance of Tabu Search strongly depends on the choice of its components: neighbourhood type, and tabu list implementation and configuration. One of the most popular choices, followed in PMTS, is the neighbourhood generated by transpositions, i.e. all solutions that can be obtained by swapping any two elements in the original permutation. The tabu list is implemented as a fixed-length list of the most recent moves (pairs of positions), which are forbidden in the next iteration, unless they lead to a solution better than the best one found so far.

There are $n \cdot (n-1)/2 = \mathcal{O}(n^2)$ permutations in a neighbourhood. In the general case, each one requires $\mathcal{O}(n^2)$ time to be evaluated, therefore the total cost of neighbourhood evaluation is $\mathcal{O}(n^4)$. In the special case of a neighbourhood generated by transpositions, the *delta function* proposed by Burkard and Rendl (1984) allows us to reduce the cost of a single solution evaluation to $\mathcal{O}(n)$, effectively reducing the cost of the neighbourhood evaluation to $\mathcal{O}(n^3)$.

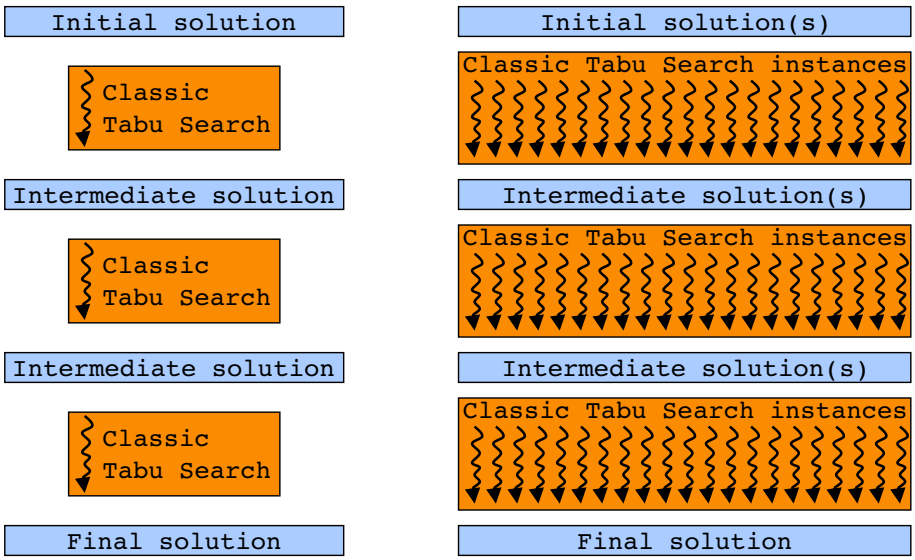
The neighbourhood evaluation is the most time-consuming part of the algorithm, since other steps of Tabu Search (move selection, tabu list management) require at most $\mathcal{O}(n)$ operations. This observation is confirmed by results from profiling — evaluation takes 80–95% of the execution time depending on the problem size.

Some research on the GPU implementations of neighbourhood evaluation has been done by Luong et al. (2009). The main disadvantage of their approach is that it requires memory transfers between the CPU and the GPU in every iteration, effectively degrading the performance. In Subsection 5.3.2, Tabu Search implemented entirely on the GPU is proposed.

5.2.2 Multistart Tabu Search Scheme

The technique of restarting a meta-heuristic with different initial solutions and different configurations proved to be effective and was successfully applied to many optimisation problems, including the recent Multistart Tabu Search (MTS) for QAP

by James et al. (2009). Due to the sequential nature of CPUs, usually just one meta-heuristic instance is run at a time, then its configuration is adjusted and the meta-heuristic is restarted (Figure 5.1 A). Multi-core CPUs and many-core modern GPUs allow several meta-heuristic instances to run at the same time (Figure 5.1 B). The latter approach will be referred to as Parallel Multistart Tabu Search (PMTS).



(A) Multistart Tabu Search (B) Parallel Multistart Tabu Search

FIGURE 5.1: Multistart Tabu Search and Parallel Multistart Tabu Search schemes.

A basic GPU implementation following the PMTS approach (SIMD-TS) was proposed by Zhu et al. (2010). In SIMD-TS, each GPU thread executed its own Tabu Search instance, and there was no communication between the threads, even at intermediate points. To diversify results, certain randomness was added to the tabu list length, and random diversification action every certain number of iterations. In Subsection 5.3.3, PMTS with two levels of parallelisation is proposed: concurrent execution of Tabu Search instances, and parallel evaluation of each neighbourhood.

5.2.3 Parallel Multistart Tabu Search for QAP

Based on the PMTS scheme presented in the previous subsection, a new algorithm for QAP is proposed. It employs a diversification technique presented in Algorithm

5.1, originally proposed by Glover (1998). The method proved to be effective when applied to Tabu Search for QAP (James et al., 2009).

Algorithm 5.1 C++ code for the diversification technique proposed by Glover (1998)

```
void Diversify(size_t offset, size_t n, const int* initial_solution,
               int* new_solution) {
  size_t pos = 0;
  for (size_t start = offset - 1; start-- > 0; ) {
    for (size_t idx = start; idx < n; idx += offset) {
      new_solution[pos++] = initial_solution[idx];
    }
  }
}
```

Tabu Search execution time depends on dataset size and number of iterations. The tabu list length and initial solution have negligible impact on the execution time, but they can have significant influence on the quality of solutions. Therefore, a model is proposed in which several Tabu Search instances are executed for the same number of iterations with different initial solutions and tabu list lengths.

For every initial solution, a set of solutions is generated using the diversification technique with *offset* from 1 to *max offset* (an algorithm parameter). For *offset* equal to 1, diversification returns the original permutation. Then, concurrent Tabu Search instances are run for R iterations for all the diversified solutions and all *tabu list lengths* provided as an algorithm parameter. For every initial solution and every tabu list length, the best solution is chosen, and then *new solutions count* (an algorithm parameter) best solutions that improved the initial solution are selected for the next generation. If no solution among those selected is better than the initial solution, the number of Tabu Search iterations is increased by a factor of $\sqrt{R}$ until it reaches *the maximum number of iterations* provided as an algorithm parameter.

5.3 Implementation Details

In this section, details of the GPU implementation of the Parallel Multistart Tabu Search are presented. Four approaches to the neighbourhood evaluation are de-

scribed in Subsection 5.3.1, followed by GPU implementation details for other steps of classic Tabu Search (Subsection 5.3.2), and the PMTS scheme (Subsection 5.3.3).

5.3.1 Neighbourhood Evaluation on CUDA

Since the neighbourhood evaluation is the most time-consuming task in the Tabu Search method, it is important that threads and memory management is done as effectively as possible. While the former is straightforward, the latter causes significant problems which are discussed below.

While each permutation in the neighbourhood can be evaluated independently, one CUDA thread is assigned to each possible swap (Figure 5.2). This way, using the delta function proposed by Burkard and Rendl (1984), neighbourhood evaluation can be performed in $\mathcal{O}(n)$ steps using $n \cdot (n-1)/2$ threads executed in parallel. In the CUDA model, threads are grouped into blocks — initial experiments showed that running 192 threads per block is optimal. This agrees with results published by Zhu et al. (2010).

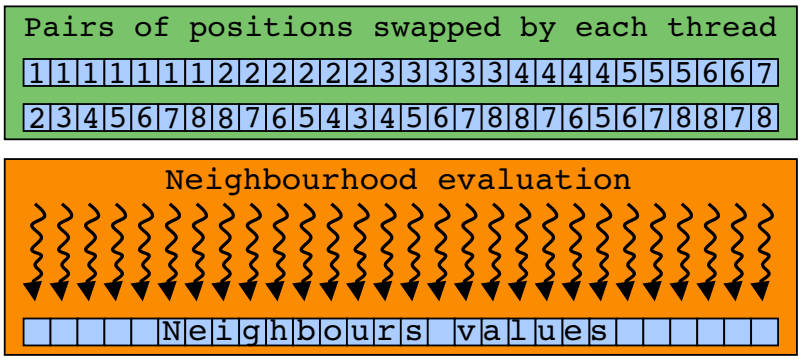


Figure 5.2: Evaluation scheme on the GPU ensuring the maximum level of coalescing.

To evaluate a neighbourhood, distance and flow matrices, and the current permutation are required. The former are the same for every thread, thus only one copy is held in the global memory. During evaluation they are accessed frequently and in irregular order depending on the current permutation. The current permutation is small enough ($4n$ bytes) to fit into fast, on-chip shared memory, and indeed it is read (with full coalescing) from global memory before evaluation begins.

If the distance and flow matrices of a QAP instance are symmetric, it is called a *symmetric instance*. In real-life applications, both symmetric and non-symmetric QAP instances can be encountered. If a QAP instance is symmetric, the number of operations, and more importantly, the number of memory reads, can be reduced by roughly 50%. This property was taken into account during implementation of CPU and GPU versions of the neighbourhood evaluation.

Typically, distance and flow matrices are too large to fit into shared memory. During the evaluation of a symmetric instance, two rows *or* two columns involved in a swap have to be read from each matrix. Values in the former matrix can be read in any order, however the current permutation determines the order in which elements are read from the latter one. If matrices are stored row-wise, and elements in *columns* are accessed in the sequential order, reads from the distance matrix can be performed in a fully coalesced way. Coalescing will occur only if the corresponding first and second columns of the distance matrix accessed by adjacent threads are close to each other. To ensure that is the case, swap moves are assigned to threads in the way presented in Figure 5.2. If rows were read instead, then no coalescing would occur.

In the case of non-symmetric instances, two rows *and* two columns from each matrix have to be read. Therefore, coalescing occurs when reading columns, but not when reading rows. Therefore, transposed copies of distance and flow matrices are stored to benefit from full coalescing — when they are stored column-wise, rows can be accessed in a fully coalesced way. Storing copies of matrices increases the memory footprint of the algorithm, but even for large datasets ($n = 256$) this is roughly an additional 0.5 MB, which can easily fit into modern GPU global memory.

In the case of the flow matrix, there is no way of ensuring that columns accessed by adjacent threads are close enough to benefit from coalescing. This is not a big issue for smaller instances, but for larger ones a decrease in performance is to be expected. Indeed, initial experiments showed that for large datasets ($n = 256$) the neighbourhood evaluation for the random permutation is roughly 2.5 times slower than the neighbourhood evaluation for the identity permutation — in the latter case the flow matrix is accessed in almost the same way as the distance matrix, i.e. reads are almost fully coalesced.

Since distance and flow matrices are not changing during runtime, another idea was to store them in a cached *constant memory* of the GPU. However, constant memory

is limited to 64 KB and the physical cache is only 8 KB in size. In consequence, only datasets with $n \leq 32$ would fit into physical cache (two 32×32 matrices with 32-bit integers = 8,128 bytes), and datasets with more than 90 facilities would not fit into constant memory at all (two 90×90 matrices = 64,800 bytes). Preliminary results showed that for $n \leq 32$ the performance of constant memory implementation is similar to the one using global memory directly. When $32 < n \leq 90$ the performance is significantly reduced due to the increasing constant memory cache miss ratio.

5.3.2 Classic Tabu Search on CUDA

Apart from the neighbourhood evaluation, two more tasks require consideration in the GPU implementation of Tabu Search: tabu list management and move selection. In the previous work we proposed GPU implementations of these tasks for Tabu Search for Permutation Flowshop Problem (Czapiński and Barnes, 2011), that can be adapted in the algorithm proposed in this chapter.

The tabu list is implemented as an array of integers, one for each possible move. A value equal to zero indicates that the corresponding move is not on the tabu list, while positive values indicate for how many iterations the corresponding move stays on the tabu list. Values of solutions obtained by moves on the tabu list are set to the maximum integer value, unless they are better than the best solution found so far. This way they will not be chosen during the move selection phase.

Move selection consists of finding the move with the lowest value, updating the current solution, and, if necessary, the best solution found so far. Finding the minimum using the tournament method and multiple processing units requires $\mathcal{O}(\log n)$ steps. This straightforward approach cannot be used, as the number of CUDA threads per block is limited, and there is no mechanism for communication between the blocks. Instead, the values array has to be divided into blocks, and the minimum is found in each block using the classic tournament method. Then, this method is recursively applied to an array of minima from each block until there is only one value left.

5.3.3 Parallel Multistart Tabu Search on CUDA

Running concurrent Tabu Search instances requires additional memory and adjusting indexing in the following operations: the neighbourhood evaluation, the tabu list

management, and the move selection. Distance and flow matrices, their transposed copies for evaluation of non-symmetric instances, and positions of swaps assigned to each thread are the same for all Tabu Search instances. However, each instance requires its own copy of:

- the current permutation,
- best solution found so far,
- an array to store values of permutations in the neighbourhood,
- an array to store the tabu list,
- two auxiliary arrays for finding the best move in the neighbourhood.

Once all Tabu Search instances are executed, the best solutions are copied from the GPU, and new configurations are chosen on the CPU (cf. Subsection 5.2.3).

5.4 Numerical Experiments

Computational experiments were conducted on a machine equipped with an Intel Core i7-980x CPU (cf. Table B.1) with 12 GB of memory, and a GeForce GTX 480 graphics card (cf. Table B.2), running Ubuntu 10.04 (64-bit). All algorithms were implemented in C++ using CUDA Toolkit 3.2. Datasets used in the performance experiments were randomly generated following the method proposed by Taillard (1991). The benchmark suite consisted of three symmetric and non-symmetric instances of each size $n \in \{8, 16, 24, \ldots, 384\}$. Datasets used in the experiments focused on solutions quality came from QAPLIB (Burkard et al., 1991). The latter benchmark consists of 137 datasets of sizes between 10 and 256.

5.4.1 CPU Implementation Details

Since the Parallel Multistart Tabu Search for the Quadratic Assignment Problem is a novel approach there are no state-of-the-art implementations that could be used for benchmarking. The CPU implementation used in the experiments includes all algorithmic optimisations described in Section 5.3. Furthermore, extra care was taken in order to optimise data locality to maximise the cache hit ratio.

The parallel CPU implementation of PMTS was prepared using MPICH2 1.2.1p1. The experiments were run on a single multi-core CPU only, but the Nemesis Communication System included in the selected MPI implementation allows for efficient intra-node communication via shared memory. Indeed, the observed parallel performance was close to the linear speed-up and was matching the performance of the corresponding OpenMP implementation. Since communication between CPU threads is infrequent, high parallel efficiency is expected in multi-CPU clusters.

5.4.2 Performance of the Neighbourhood Evaluation on the GPU

Computational experiments were conducted in order to measure the performance of the neighbourhood evaluation and compare it against the single-core CPU implementation. All four evaluation methods presented in Subsection 5.3.1 were considered. Wall-clock execution times were measured in ten independent runs, averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5% (less than 1% in most cases). Speed-ups were calculated according to the formula

$$S = \frac{T_{CPU}}{T_{GPU}}, \tag{5.2}$$

where T_{CPU} and T_{GPU} denote the execution time on the corresponding platform.

The performance of all evaluation methods increased quickly with the problem size (Figure 5.3). The increase was connected with better GPU utilisation for larger instances. Methods with no, or partial coalescing, reached the peak performance (speed-ups by a factor of 40 for symmetric, and 70 for non-symmetric instances) for $n = 96$. When $n > 96$, speed-ups did not change significantly and stayed at around a factor of 30 for symmetric, and 50 for non-symmetric instances.

Peak performance for methods with full coalescing was achieved for larger instances (Figure 5.3), but speed-ups were significantly better — 240 times for $n = 200$ (symmetric) and 300 times for $n = 160$ (non-symmetric). Then, speed-ups decreased with the increasing problem size and reached a factor of 150 for $n = 384$. This decrease in performance is connected with the increasing number of non-coalesced reads from the flow matrix — only reads from the distance matrix can be fully coalesced (cf. Subsection 5.3.1).

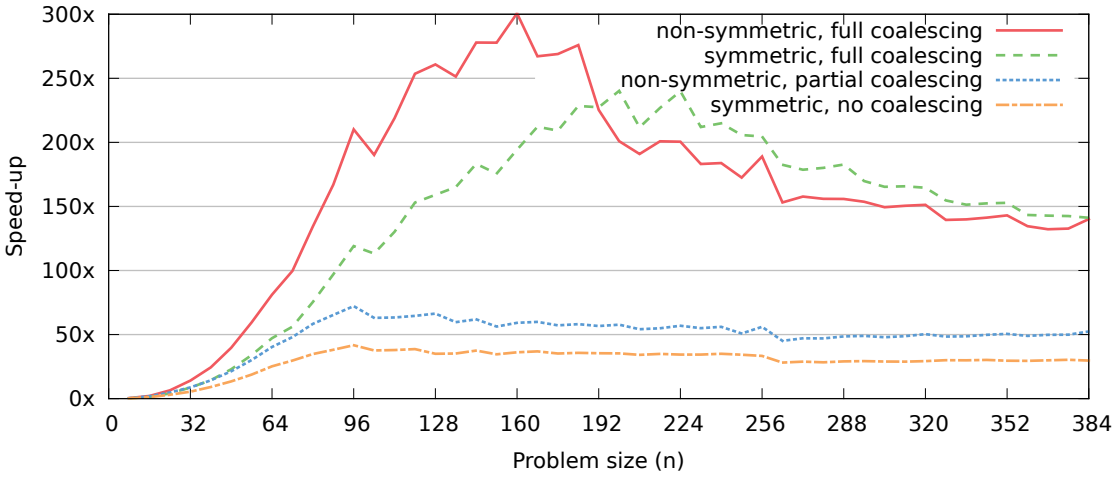


Figure 5.3: Performance of neighbourhood evaluation methods. Speed-ups were measured against the CPU code running on a single core of Intel i7-980x. Ordering the data to maximise the fraction of coalesced global memory accesses is crucial for the performance.

The difference in speed-ups between symmetric and non-symmetric instances was caused by a more cache-friendly memory access pattern in the CPU implementation for symmetric datasets. Interestingly, speed-ups of coalesced evaluation methods for symmetric and non-symmetric problems converged for large datasets (Figure 5.3).

5.4.3 Performance of Parallel Multistart Tabu Search on the GPU

Computational experiments were conducted in order to measure parallel efficiency of the GPU implementation of Parallel Multistart Tabu Search. The parallel CPU (MPI) implementation was used as a reference. Each algorithm had to execute 30 Tabu Search instances being limited to run a fixed number of concurrent Tabu Search instances, denoted by m. The CPU implementation was using 1–6 cores and respective m values; the CUDA implementation was run for $m \in \{1, 2, \ldots, 30\}$.

Algorithmic speed-ups for both implementations were computed using the classic formula $T(1)/T(m)$, where $T(m)$ denotes the execution time using m concurrent Tabu Search instances. In addition, CUDA implementation speed-ups were also measured against an MPI counterpart run on all six cores ($m = 6$).

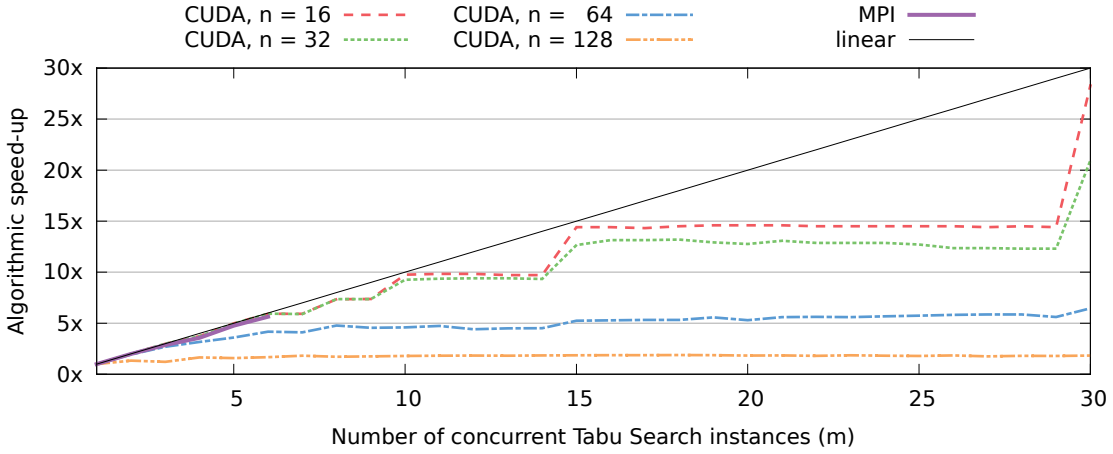
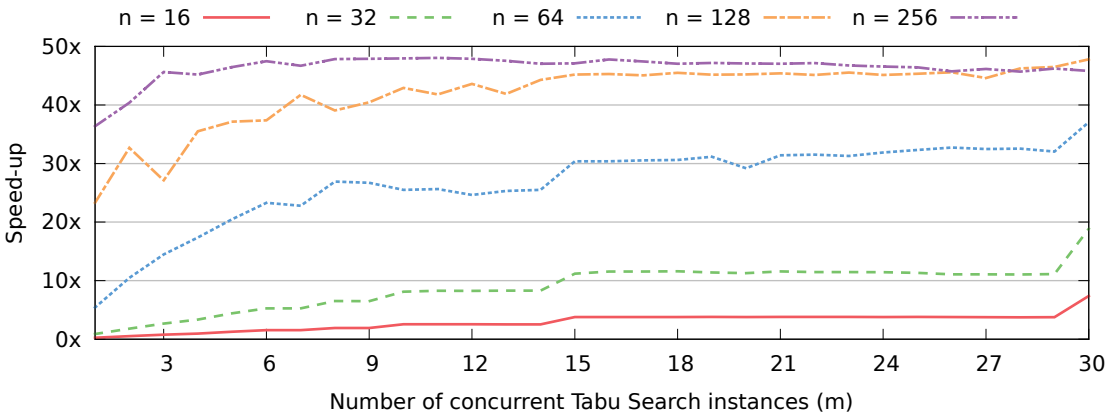


Figure 5.4: Algorithmic speed-ups of MPI and CUDA implementations of PMTS (n denotes problem size). Results are the same for symmetric and non-symmetric instances.

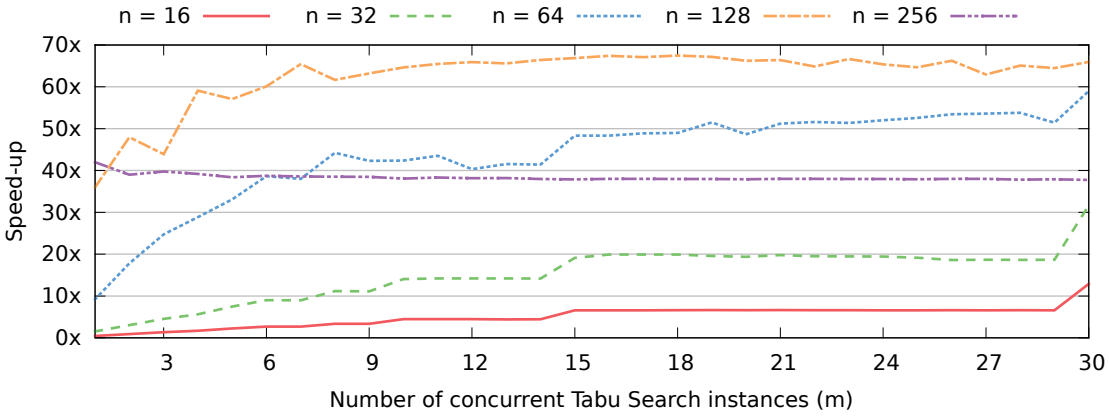
For small instances, algorithmic speed-ups were very close to linear for $m \leq 6$ (Figure 5.4). Beyond that point, the CUDA implementation experienced performance degradation due to idle cores whenever m was not a factor of 30, e.g. for $m = 14$ two batches of 14 Tabu Search instances were executed, and then one batch with just two instances. Furthermore, there was an overhead connected with running each batch — initial solutions had to be transferred to the GPU memory and then the results had to be copied back. The fraction of this overhead in total execution time was higher for smaller datasets. Indeed, for instances with $n \in \{16, 32\}$ every time the number of batches decreased, a noticeable increase in speed-up could be observed while for $n = 128$ this phenomenon was not observed at all.

Algorithmic speed-ups of CUDA implementation increased with m only to the point when the GPU was fully utilised (Figure 5.4) — the actual computational capabilities of the GPU were not growing. For $n = 16$ this point was not reached even for $m = 30$ ($n(n-1)/2$ threads are run per Tabu Search instance; while blocks consisted of 192 threads, 30 blocks were executed), while for $n = 128$ it was reached for just one Tabu Search instance (43 blocks). This phenomenon could not be observed for the MPI implementation, as here the computational resources (CPU cores) were growing with increasing number of concurrent Tabu Search instances.

CUDA speed-ups against the MPI implementation increased quickly with m values to the point when the GPU is fully utilised, and then remained stable (Figure



(A) Symmetric instances

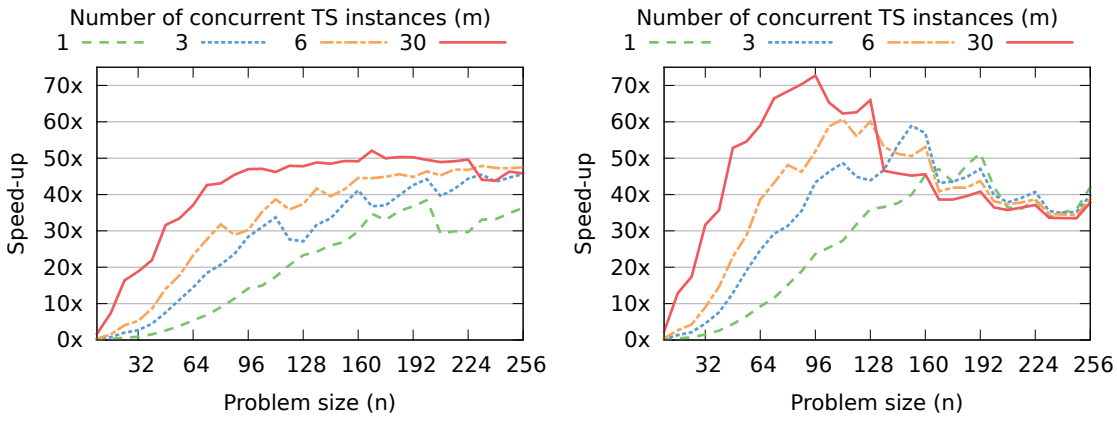


(B) Non-symmetric instances

FIGURE 5.5: PMTS performance for different problem sizes. Speed-ups were measured against the CPU implementation running on six cores of an Intel i7-980x.

5.5, $n \in \{128, 256\}$). Again, the significant increase in performance whenever the number of batches decreased was connected with smaller batch execution overheads and could be observed for smaller instances ($n \leq 64$). In general, better speed-ups could be observed for larger datasets except for very large ($n = 256$) non-symmetric instances. This was connected with the neighbourhood evaluation performance — peak performance for non-symmetric instances could be observed for $n \in [144, 184]$, or $n \in [176, 256]$ for symmetric instances (Figure 5.3).

Running concurrent Tabu Search instances significantly increased PMTS speed-ups for smaller instances (Figure 5.6). The differences decreased for larger problems due to better GPU utilisation. The performance converged for non-symmetric instances



(A) Symmetric instances (B) Non-symmetric instances

FIGURE 5.6: PMTS performance for different numbers of concurrent Tabu Search instances. Speed-ups were measured against the CPU implementation running on six cores of an Intel i7-980x. Differences with respect to m are connected with the utilisation of the GPU, and tend to disappear for sufficiently large problems. Higher speed-ups for smaller non-symmetric problems (B) are connected with the lower cache hit ratio on the CPU.

($n \geq 160$). The convergence was not achieved for symmetric instances even for $n = 256$, but differences were noticeably smaller than for small instances.

For sufficiently large grids, the GPU implementation of the PMTS method is 50 times (symmetric instances) or 70 times (non-symmetric instances) faster than the CPU implementation (Figure 5.6).

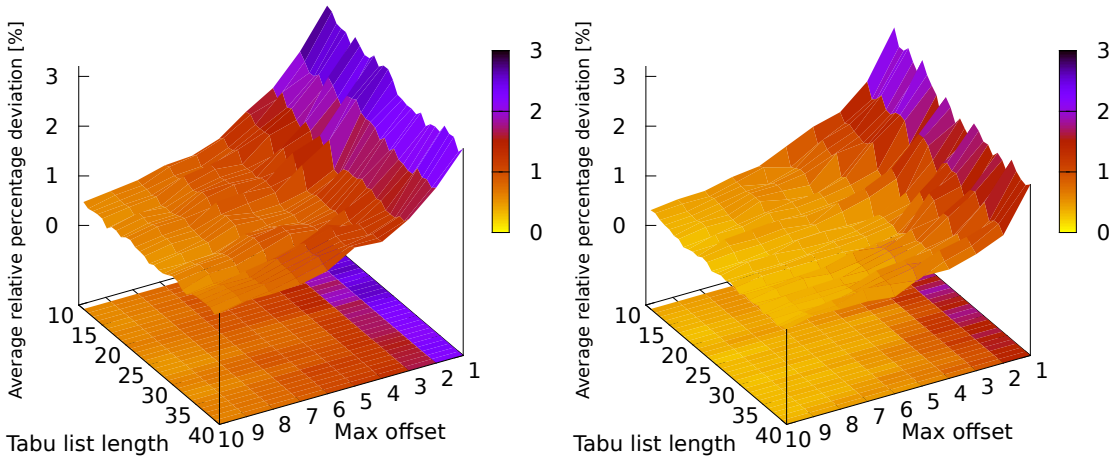
5.4.4 The Quality of Solutions Obtained by PMTS for QAP

Initial experiments were carried out on the single-configuration PMTS, i.e. only one tabu list length and only one solution was used in the next generation. PMTS was run with tabu list lengths between 16 and 40, *max offset* between 1 and 10, and the maximum number of Tabu Search iterations set to $n^{1.5}$ and n^2. Then, the relative percentage deviation (RPD) defined as

$$\frac{C(\pi) - C(\pi^*)}{C(\pi^*)} \cdot 100\% \tag{5.3}$$

was calculated and averaged (ARPD) for all 137 datasets. Here, $C(\pi)$ denotes the cost of solution returned by PMTS and $C(\pi^*)$ the cost of the optimal (or the

best known) solution. Lower RPD values indicate better solutions; negative values indicate that the new best solution was found.



(A) $n^{1.5}$ iterations (B) n^2 iterations

FIGURE 5.7: The quality of solutions returned by the single-configuration PMTS (in terms of ARPD, lower is better). Increasing *max offset* results in better quality solutions at the cost of longer execution time. The tabu list length does not affect the execution time, but it can significantly impact the quality of solution. However, there is no way to predict optimal tabu list length for a particular problem.

The quality of solutions was increasing with *max offset* values (Figure 5.7), but at the cost of increasing execution time. Improvement in quality was fast for *max offset* values up to five, making the choice of *max offset* $= 5$ a compromise between execution time and solutions quality. Further improvement with increasing *max offset* was slower but steady. Moreover, the solutions obtained by runs with the maximum number of iterations equal to n^2 (Figure 5.7B) were of better quality than those obtained with the maximum number of iterations equal to $n^{1.5}$ (Figure 5.7A). Again, this improvement came at the cost of a longer execution time.

There was no clear correlation between the tabu list length and the quality of solutions. Indeed, for every tabu list length the quality of solutions varied significantly for different datasets. To overcome the optimal tabu list length choice problem, PMTS was run with multiple tabu list lengths at the same time.

Tabu list lengths were chosen based on the single-configuration results. Sets with five tabu list lengths were $\{18, 20, 22, 28, 30\}$ for $n^{1.5}$ iterations and $\{18, 21, 24, 29, 32\}$ for

n^2 iterations, and the set with ten lengths was $\{18, 19, 20, 21, 24, 26, 29, 30, 32, 35\}$ for $n^{1.5}$ and n^2 iterations. The number of the best solutions used in the next generation was increased from one to five. In this experiment 134 datasets from QAPLIB with up to 128 facilities were considered.

The quality of solutions increased monotonically with the number of iterations, the maximum offset, and the number of tabu list lengths (Figure 5.8). The number of the best known solutions found followed the same trend, but was not as consistent — for n^2 iterations, the configuration with *max offset* = 10 and five tabu list lengths found more best known solutions, than two configurations that required more time. This was connected with the influence of the tabu list length on the quality of solutions for particular datasets.

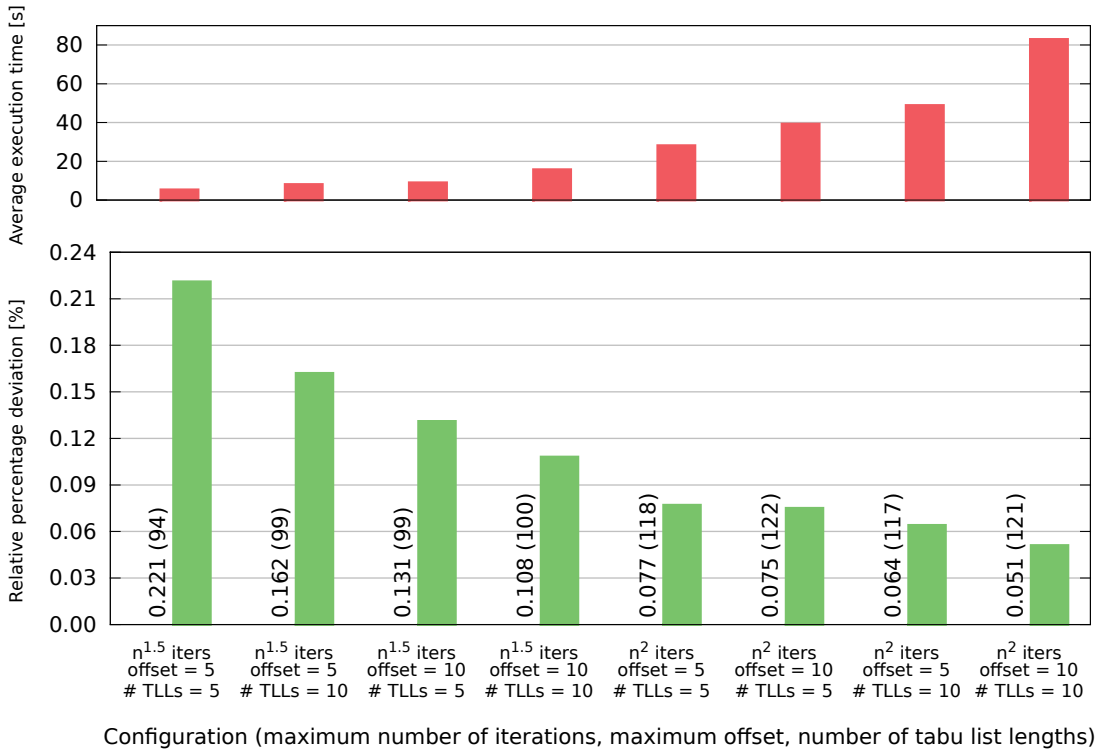


Figure 5.8: The quality of solutions returned by the multi-configuration PMTS. The number of the best known solutions (out of 134) found is given in the brackets.

The increase in the quality of solutions observed for more demanding configurations came at the price of a longer execution time (Figure 5.8). However, even for the

Table 5.1: Results for different configurations of PMTS for `sko*` datasets from the QAPLIB benchmark suite (Burkard et al., 1991).

		$n^{1.5}$ iterations $max\ offset = 5$ $\#\ TLLs = 5$		n^2 iterations $max\ offset = 5$ $\#\ TLLs = 5$		n^2 iterations $max\ offset = 10$ $\#\ TLLs = 10$	
Dataset	Best known solution	RPD [%]	Time [s]	RPD [%]	Time [s]	RPD [%]	Time [s]
`sko42`	15,812	0.000	0.5	0.000	3	0.000	9
`sko49`	23,386	0.068	1.5	0.000	6	0.000	17
`sko56`	34,458	0.012	2.6	0.000	17	0.000	42
`sko64`	48,498	0.054	4.9	0.000	13	0.000	36
`sko72`	66,256	0.160	6.4	0.000	63	0.000	170
`sko81`	90,998	0.062	10.1	0.033	60	0.000	287
`sko90`	115,534	0.196	37.6	0.000	355	0.000	381
`sko100a`	152,002	0.092	37.8	0.000	232	0.000	525
`sko100b`	153,890	0.043	40.0	0.000	197	0.000	1015
`sko100c`	147,862	0.073	37.7	0.000	461	0.001	1008
`sko100d`	149,576	0.126	55.6	0.066	247	0.000	742
`sko100e`	149,150	0.009	54.8	0.000	226	0.004	963
`sko100f`	149,036	0.121	57.2	0.000	290	0.023	669
Average		**0.078**	**26.7**	**0.008**	**167**	**0.002**	**451**

largest datasets and the most time-consuming configuration, PMTS executed in less than 17 minutes (Table 5.1). Significant differences in execution times for datasets of the same size (e.g. `sko100a` and `sko100b`; Burkard et al., 1991) were observed because PMTS does not perform a fixed number of iterations, but rather adjusts it according to the quality of intermediate solutions. Three of the best known solutions found by the second configuration were not found by the third, more time-consuming one, but still the average relative percentage deviation was decreasing (results were improving) with the increasing execution time.

The results obtained in experiments show that PMTS is capable of finding good quality results in a short time — for the simplest multi-configuration, PMTS found the best known solutions for 94 out of 134 benchmark datasets. On average, the results were only 0.22% worse than the best known solutions, and even for the largest datasets they were found in less than a minute. Increasing configuration parameters (maximum number of iterations or offset), or including more tabu list lengths, resulted in further monotonic improvement in the quality of the results.

5.5 Conclusions

- This chapter presents a novel Parallel Multistart Tabu Search (PMTS) for the Quadratic Assignment Problem implemented using CUDA. PMTS is based on the multiple Tabu Search instances running in parallel entirely on the GPU.

- PMTS runs up to 50 times faster for symmetric instances, and up to 70 times faster for non-symmetric instances, in comparison to the multi-threaded implementation running on six cores of a modern, high-end CPU.

- This outstanding performance is achieved due to parallelised neighbourhood evaluation, and careful memory layout and access patterns maximising the fraction of coalesced global memory reads. Moreover, the first GPU implementation of the tabu list data structure was used. This allows the Tabu Search to run entirely on the GPU and avoid overheads connected with the CPU-GPU communication.

- In terms of the quality of solutions, PMTS is capable of delivering good quality (often optimal or the best known) solutions in a short time. PMTS is highly customisable — a more thorough search, resulting in further improvement in the quality of solutions, is possible at the cost of a longer execution time.

Part II
Distributed Block Direct Solver

This part of the thesis presents the research on development of the Distributed Block Direct Solver (DBDS). The DBDS method is a novel distributed solver applicable to a wide range of linear systems arising from the discretisation of partial differential equations. Furthermore, the solver is designed to work on distributed memory multi-GPU platforms. The Distributed Block Direct Solver was motivated by a recently published Distributive Conjugate Gradients (DCG) method (Becker, 2006) and follows the same Additive Schwarz decomposition method.

The DCG method is an adaptation of the Preconditioned Conjugate Gradients algorithm, therefore Chapter 6 is focused on efficient GPU implementations of PCG with a range of preconditioners. That chapter also contains an overview of the existing CPU and GPU libraries offering PCG functionality. Thanks to careful optimisations, the performance of the GPU implementation proposed in Chapter 6 is more than 50% faster than a state-of-the-art solver from the CUSP library.

Chapter 7 introduces the Distributed Block Direct Solver. DCG and DBDS methods divide the original system into smaller subproblems that can be solved almost independently. Since matrices in local linear systems are fixed and only the right-hand side vectors are changing between the iterations, a new idea in DBDS was to replace PCG-style iterations in DCG with a direct sparse solver. The class of problems for which the DBDS method converges was identified, and the procedure to calculate a high quality upper bound on the convergence rate was derived. Finally, the possible extensions to the DBDS method are discussed: subdomain overlapping and mixed-precision iterative improvement.

Chapter 8 presents the results of numerical experiments on the DCG and DBDS methods on distributed memory parallel platforms: Cranfield University supercomputer Astral, and a multi-GPU cluster. A direct sparse solver is the main component of DBDS, therefore a range of available implementations was compared to find the fastest one. Then, the performance of hybrid DCG and DBDS solvers was investigated. Experiments were also conducted on performance improvements from extensions to the DBDS method, and on scalability and time-effectiveness of DCG and DBDS on a large-scale CPU cluster with Infiniband interconnect.

Chapter 6

Preconditioned Conjugate Gradients on the GPU

6.1 Introduction

The Preconditioned Conjugate Gradients (PCG) algorithm is among the most effective methods for finding a solution to a sparse linear system. Many implementations have been proposed over the years, including GPU solutions (Buatois et al., 2009; Stock and Koch, 2010; Wozniak et al., 2010). This chapter introduces an optimised GPU implementation of the PCG method with various preconditioners: the Diagonal Preconditioner, Incomplete Cholesky and LU Preconditioners, and the Polynomial Preconditioner.

The PCG solver is part of the High-performance Parallel Solvers Library (HPSL), which was developed throughout this project. Furthermore, an in-depth analysis of the PCG performance is provided. The proposed implementation is able to achieve 50–68% of the theoretical peak performance on modern GPUs — 50% faster than a state-of-the-art implementation in the CUSP library (Bell and Garland, 2012).

In 2006, Becker proposed the Distributive Conjugate Gradients (DCG) method, which parallelises PCG execution in distributed memory systems. The conclusions from this chapter played a crucial role in developing a high-performance implementation of the DCG algorithm for GPU platforms, and development of the Distributed

Block Direct Solver — a novel approach to the solution of sparse linear systems in hybrid CPU-GPU parallel systems (Chapter 8).

This chapter is organised as follows. The remaining part of this section provides a review of existing libraries offering Conjugate Gradients functionality on the CPU (Subsection 6.1.1) and on the GPU (Subsection 6.1.2). Section 6.2 describes the PCG method and various preconditioners, and offers the time complexity analysis. Section 6.3 provides details on how the PCG method is implemented in the HPSL library. Section 6.4 presents the results and discussion of the numerical experiments: an in-depth analysis of the PCG performance, and comparison of time-effectiveness of various preconditioners. Finally, the conclusions are given in Section 6.5.

6.1.1 Existing CPU Solutions

Numerous numerical libraries providing the Conjugate Gradients method have been developed over the years. The most prominent implementations are briefly discussed below. A more comprehensive overview is provided by Dongarra and Gates (2013).

The available libraries differ significantly on the level of abstraction, target architecture, and the performance. The low-level building blocks (e.g. BLAS, Blackford et al., 2002) offer the best flexibility, but at the cost of more difficult development. On the other side, there are libraries that reduce running the Conjugate Gradients solver to a single call, hiding all the details from the user. The trade-off for using high-level abstractions is less ability to tune up, and in consequence, generally lower performance. Typically, the low-level libraries can be fine-tuned to run faster, especially in environments enabling parallel processing.

There is a whole range of BLAS implementations on the CPU varying in performance from the relatively slow reference Netlib implementation (`http://www.netlib.org/blas/`) to the fastest Intel MKL (Math Kernel Library, `http://software.intel.com/en-us/intel-mkl/`), which is multi-threaded and highly optimised for most Intel CPUs including the recent Intel Xeon Phi co-processors (Jeffers and Reinders, 2013). ATLAS (Automatically Tuned Linear Algebra Software, Whaley et al., 2001; Whaley and Petitet, 2005) is another BLAS implementation, notable for auto-tuning ability. Even though ATLAS is not as fast as Intel MKL, it is available under an open-source licence, whereas Intel MKL is an expensive proprietary technology.

While some BLAS implementations support multi-threading, none of them are capable of utilising distributed memory parallel systems. This issue is addressed by the Portable, Extensible Toolkit for Scientific Computation library (PETSc, Balay et al., 1997, 2013a,b). PETSc supports MPI and shared memory pthreads. Furthermore, the latest versions also support computation on the GPUs. However, PETSc still uses a low-level BLAS-like interface, which makes development difficult.

A higher level of abstraction is offered by the open source Eigen library (Guennebaud et al., 2010). Thanks to C++ templates, Eigen enables MATLAB-like syntax, maintaining high performance at the same time. In addition, Eigen applications are easily portable (no need for library binary, since the required Eigen codes are compiled from the source) and has a simple support for switching between single and double precision arithmetic, and between row- and column-major matrix data layout.

Finally, the highest level of abstraction is offered by the Trilinos Project (Heroux et al., 2003, 2005). It is similar to PETSc in that both libraries use distributed matrices and vectors, however Trilinos has an explicit modular architecture where independent components are designed to work with each other. The Conjugate Gradients solver is part of the Belos package and can be used in a black-box manner. The Eigen library offers similar functionality, however this chapter aims to provide a detailed performance analysis, therefore it was decided not to use such solvers.

6.1.2 Existing GPU Solutions

General purpose computing on the GPU (GPGPU) is still a relatively new area of research. In consequence, the choice of GPU libraries for linear algebra is not as broad as on CPUs. However, GPU codes for BLAS operations and high-level numerical solvers are available.

For basic dense linear algebra operations, NVIDIA offers the CUBLAS library (NVIDIA, 2012a). Alternatively, BLAS and LAPACK routines on the GPU are also implemented in the Matrix Algebra on GPU and Multi-core Architectures library (MAGMA, `http://icl.utk.edu/magma/`).

However, the Conjugate Gradients method works best for sparse linear systems and requires efficient matrix-vector multiplication. This is provided by CUSPARSE

(NVIDIA, 2014) — a library for sparse linear algebra. CUSPARSE is compatible with CUBLAS and supports a range of matrix storage formats and operations: sparse matrix-vector multiplication, triangular and tridiagonal systems solvers, and incomplete LU and Cholesky factorisations which can be used as preconditioners.

The black-box implementation of the Conjugate Gradients solver is available on the GPUs thanks to the CUSP library (Bell and Garland, 2012), which provides generic parallel algorithms for sparse matrix and graph computations. The entire CG solver can be executed in a single line of code, however there is little control on algorithm execution, which makes it unsuitable for a detailed analysis.

6.2 Preconditioned Conjugate Gradients

The Conjugate Gradients (CG) algorithm (Hestenes and Stiefel, 1952) is one of the fastest and most popular iterative methods used to solve large sparse linear systems. In each iteration, the solution vector $\mathbf{x}$ is adjusted to minimise the residual vector $\mathbf{r}$ along a certain direction vector $\mathbf{p}$ (Algorithm 6.1). By making sure that all the directions are mutually *conjugate*, the convergence of the CG method is guaranteed, but only for symmetric and positive-definite (SPD) linear systems. For more details refer to Subsection 2.2.3 in the literature review chapter.

Algorithm 6.1 The Conjugate Gradients method

1: $\mathbf{p}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
2: **for** $r = 0, 1, \ldots$ **do**
3: $\quad \alpha^{(r)} = (\mathbf{r}^{(r)}, \mathbf{r}^{(r)}) / (\mathbf{A}\mathbf{p}^{(r)}, \mathbf{p}^{(r)})$
4: $\quad \mathbf{x}^{(r+1)} = \mathbf{x}^{(r)} + \alpha^{(r)} \mathbf{p}^{(r)}$
5: $\quad \mathbf{r}^{(r+1)} = \mathbf{r}^{(r)} - \alpha^{(r)} \cdot \mathbf{A}\mathbf{p}^{(r)}$
6: $\quad \beta^{(r)} = (\mathbf{r}^{(r+1)}, \mathbf{r}^{(r+1)}) / (\mathbf{r}^{(r)}, \mathbf{r}^{(r)})$
7: $\quad \mathbf{p}^{(r+1)} = \mathbf{r}^{(r+1)} + \beta^{(r)} \mathbf{p}^{(r)}$
8: **end for**

6.2.1 Time Complexity

Each iteration of the Conjugate Gradients method consists of a sparse matrix-vector multiplication (SpMV), dot products, and vector additions (with optional

scaling, typically denoted as AXPY). All three operations have low computation-to-communication ratio, thus they can be considered memory-bound.

In SpMV, each non-zero element has to be multiplied by a corresponding vector element and added to the solution vector. Let n denote the matrix size, and nnz the number of non-zero elements, then the time complexity is $\mathcal{O}(n + nnz)$. The SpMV can be parallelised: all coefficient multiplications can be performed concurrently, then the elements in each row can be added independently. The reduction in each row can also be parallelised.

However, due to highly irregular memory access patterns, efficient SpMV implementation is not trivial on GPU platforms. Many papers on possible optimisations have been published (Bell and Garland, 2008; Kourtis et al., 2008; Baskaran and Bordawekar, 2009; Williams et al., 2009; Guo and Wang, 2010; Li et al., 2013).

The complexity of AXPY and dot product operations is linear with respect to the matrix size n. The former operation is easy to parallelise and can be completed in a constant time using n threads. The latter operation involves non-trivial data dependencies and requires $\mathcal{O}(\log n)$ steps even when n threads are used.

Typically, the number of non-zero elements nnz is likely to be many times higher than n. Since SpMV is the only step that depends on nnz, it is likely to become a dominant operation in the Conjugate Gradients solver.

6.2.2 Preconditioners

The convergence rate, i.e. the number of iterations required to reach a solution with a desired error tolerance, strongly depends on a particular linear system (Shewchuk, 1994, §9). The number of iterations can be reduced by using a *preconditioner*, i.e. implicitly modifying the original matrix in order to improve the numerical properties (spectral radius) and, in consequence, the convergence rate (Barrett et al., 1994, Ch. 3). This enhanced algorithm is called the Preconditioned Conjugate Gradients method and is presented in Algorithm 6.2 (matrix $\mathbf{M}$ is called the preconditioner).

Mathematically, the preconditioner can be expressed as a linear operator $\mathbf{M}$, and the original system of equations

$$\mathbf{Ax} = \mathbf{b} \tag{6.1}$$

Algorithm 6.2 The Preconditioned Conjugate Gradients method

1: $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
2: $\mathbf{z}^{(0)} = \mathbf{M}^{-1}\mathbf{r}^{(0)}$
3: $\mathbf{p}^{(0)} = \mathbf{z}^{(0)}$
4: **for** $r = 0, 1, \ldots$ **do**
5: $\quad \alpha^{(r)} = (\mathbf{r}^{(r)}, \mathbf{z}^{(r)})/(\mathbf{A}\mathbf{p}^{(r)}, \mathbf{p}^{(r)})$
6: $\quad \mathbf{x}^{(r+1)} = \mathbf{x}^{(r)} + \alpha^{(r)}\mathbf{p}^{(r)}$
7: $\quad \mathbf{r}^{(r+1)} = \mathbf{r}^{(r)} - \alpha^{(r)} \cdot \mathbf{A}\mathbf{p}^{(r)}$
8: $\quad \mathbf{z}^{(r+1)} = \mathbf{M}^{-1}\mathbf{r}^{(i+1)}$
9: $\quad \beta^{(r)} = (\mathbf{r}^{(r+1)}, \mathbf{z}^{(r+1)})/(\mathbf{r}^{(r)}, \mathbf{z}^{(r)})$
10: $\quad \mathbf{p}^{(r+1)} = \mathbf{z}^{(r+1)} + \beta^{(r)}\mathbf{p}^{(r)}$
11: **end for**

is replaced with

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}. \tag{6.2}$$

The convergence rate of the Conjugate Gradients method depends on the condition number and the distribution of the eigenvalues of matrix $\mathbf{A}$. If $\mathbf{M}$ is a good approximation of $\mathbf{A}$, then the augmented matrix $\mathbf{M}^{-1}\mathbf{A}$ has a lower condition number and more favourable eigenvalue distribution. In consequence, the Preconditioned Conjugate Gradients method requires fewer iterations than the classic CG algorithm.

In addition to being a good approximation of matrix $\mathbf{A}$, matrix $\mathbf{M}$ should be such that computing

$$\mathbf{z} = \mathbf{M}^{-1}\mathbf{r} \tag{6.3}$$

is fast, since it is computed at each iteration (Algorithm 6.2, lines 2 and 8). This operation and the construction of $\mathbf{M}$ comprise the preconditioning overhead.

Many different preconditioners for the PCG have been proposed. In this chapter, the following four methods are considered. Again, n denotes the matrix size, and nnz the number of non-zero elements.

Diagonal Preconditioner is the simplest approach considered in this study. The operator $\mathbf{M}$ consists of diagonal elements of matrix $\mathbf{A}$, i.e. $\mathbf{M} = diag(\mathbf{A})$. The construction step involves extracting and inverting the diagonal elements, and the solution step becomes a coefficient-wise vector multiplication. Both

operations require $\mathcal{O}(n)$ time and are susceptible to parallelisation — each element can be processed independently.

Polynomial Preconditioner is based on the approximation of the inverse of the matrix $\mathbf{A}$ by a Neumann series. If $\mathbf{A} = \mathbf{I} - \mathbf{G}$ and spectral radius $\rho(\mathbf{G}) < 1$, the following relation is true (Saad, 2003, Ch. 10):

$$\sum_{k=0}^{\infty} \mathbf{G}^k = (\mathbf{I} - \mathbf{G})^{-1} = \mathbf{A}^{-1}. \tag{6.4}$$

In consequence, the inverse of matrix $\mathbf{A}$ can be approximated by using the first few elements of the series, i.e. $\mathbf{M}^{-1} \approx \sum_{k=0}^{K} \mathbf{G}^k$, where K is the degree of the Polynomial Preconditioner.

There are many variants of Polynomial Preconditioners. One of the simplest versions is described by Liang (2005, §4.2) and simply adds a scaling factor ω, i.e. $\mathbf{G} = \mathbf{I} - \omega\mathbf{A}$. In this case,

$$\sum_{k=0}^{\infty} \mathbf{G}^k = (\mathbf{I} - \mathbf{G})^{-1} = \omega^{-1}\mathbf{A}^{-1} \tag{6.5}$$

$$\mathbf{A}^{-1} \approx \omega \left(\sum_{k=0}^{K} \mathbf{G}^k \right). \tag{6.6}$$

The scaling factor can be used to ensure that the spectral radius of matrix $\mathbf{G}$ is indeed lower than one. Liang suggests to use the inverse of the largest eigenvalue, however it is often unknown and not practicable to compute. In the experiments, the inverse of the largest diagonal element was used instead.

The Polynomial Preconditioner can be further generalised by introducing another easily invertible linear operator $\mathbf{N}$. In this case $\mathbf{G} = \mathbf{I} - \mathbf{N}^{-1}\mathbf{A}$, and

$$\sum_{k=0}^{\infty} \mathbf{G}^k = (\mathbf{I} - \mathbf{G})^{-1} = \mathbf{A}^{-1}\mathbf{N} \tag{6.7}$$

$$\mathbf{A}^{-1} \approx \left(\sum_{k=0}^{K} \mathbf{G}^k \right) \mathbf{N}^{-1}. \tag{6.8}$$

Becker (2006, §7.4.4) followed this approach and used $\mathbf{N} = diag(\mathbf{A})$. Note that if all diagonal elements of matrix $\mathbf{A}$ are the same (e.g. matrix from discretisation of Poisson's Equation), both variants of the Polynomial Preconditioner will return the same results.

In the latter approach, the preconditioning operator takes the form

$$\mathbf{M}^{-1} = \left(\sum_{k=0}^{K} \left(\mathbf{I} - diag(\mathbf{A})^{-1}\mathbf{A} \right)^k \right) diag(\mathbf{A})^{-1}. \tag{6.9}$$

Pre-computing $\mathbf{M}^{-1}$ for the higher degrees K is prohibitively expensive, therefore it is not stored explicitly. Instead, the solution step $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$ is performed using matrix $\mathbf{G}$ and Horner's scheme for polynomial evaluation:

1: $\mathbf{v} = diag(\mathbf{A})^{-1}\mathbf{r}$
2: $\mathbf{z} = \mathbf{v}$
3: **for** $k = 1, 2, \ldots, K$ **do**
4: $\quad \mathbf{z} = \mathbf{G}\mathbf{z} + \mathbf{v}$
5: **end for**

Construction of the Polynomial Preconditioner consists of extracting and inverting the diagonal of matrix $\mathbf{A}$, and then computing matrix $\mathbf{G}$ — the time complexity is $\mathcal{O}(n+nnz)$. Sparse matrix-vector multiplication is the dominant operation during the preconditioner solution step, therefore the complexity is $\mathcal{O}(K(n + nnz))$. The parallelisation is similar to that of SpMV operation.

Incomplete LU Preconditioner (Saad, 2003, §10.3) is based on the decomposition of matrix $\mathbf{A}$ into triangular factors. Knowing the complete LU decomposition ($\mathbf{A} = \mathbf{LU}$) would allow us to quickly solve the original system. However, for sparse linear systems it is possible that the LU factors would consist mostly of non-zero elements and would become too large to fit into memory. Incomplete factorisation addresses this problem: only the elements corresponding to non-zero elements in the original matrix are computed. In consequence, the original matrix is approximated by a multiplication of two sparse triangular matrices, i.e. $\mathbf{A} \approx \tilde{\mathbf{L}}\tilde{\mathbf{U}}$.

The preconditioner construction is expensive and depends on the number of non-zero elements: $\mathcal{O}(nnz^2)$ operations are required. The solution phase consists of a forward and a backward substitution and requires $\mathcal{O}(nnz)$ time. Both operations can benefit from limited parallelisation, which depends on the sparsity pattern of matrix $\mathbf{A}$.

Incomplete Cholesky Preconditioner is also based on an incomplete decomposition — Cholesky factorisation, which can only be applied to SPD matrices.

The symmetry is used during the factorisation step, reducing the number of required operations by a factor of two compared to ILU. Furthermore, using Cholesky decomposition allows us to store only one triangular factor (the other is its transpose) and reduce the memory requirements by a factor of two.

6.3 Implementation Details

The codes used in the experiments are part of the High-performance Parallel Solvers Library (HPSL). The main objective was to offer a high-level abstraction and extensibility, at the same time maintaining the high performance and transparent transition between computation on the CPU and the GPU.

The existing libraries were used as a framework for HPSL development: Eigen library for the CPU codes, and CUBLAS and CUSPARSE for the GPU codes. This choice is justified in Subsection 6.3.1. Then, Subsection 6.3.2 explains the optimisations enabling the high-performance of the solvers included in the HPSL library.

6.3.1 Choosing the Library

After the initial tests on the CPU, two libraries were considered: PETSc and Eigen. PETSc was picked for its built-in support for distributed computing and BLAS-like low-level interface offering good performance. The concrete Conjugate Gradients implementation using the PETSc library is presented in Algorithm 6.3.

In addition to relatively difficult programming, the main shortcoming of PETSc is that it can only be compiled for a specific arithmetic precision. This does not allow for a mixed-precision iterative improvement approach considered in Chapter 8.

The Eigen library is much easier to program and offers a MATLAB-like interface — the example Conjugate Gradients implementation in Eigen is presented in Algorithm 6.4. Eigen uses C++ templates, which allow easy switching between single and double precision and between row- and column-major matrix storage. Eigen provides easy integration with the existing sparse direct solver libraries: LU decomposition for sparse matrices UMFPACK (`http://www.cise.ufl.edu/research/sparse/umfpack/`), SuperLU (`http://crd-legacy.lbl.gov/~xiaoye/SuperLU/`),

Algorithm 6.3 Conjugate Gradients implementation using PETSc library

```
void ConjugateGradients(size_t max_iter, Mat A, Vec b, Vec x) {
  Vec r, p, aux;
  VecDuplicate(b, &r); VecDuplicate(b, &p); VecDuplicate(b, &aux);
  VecCopy(b, r);
  VecCopy(r, p);
  VecZeroEntries(x);

  PetscScalar rr;
  VecDot(r, r, &rr);
  for (size_t iter = 0; iter < max_iter; ++iter) {
    MatMult(A, p, aux);  // aux = A * p
    PetscScalar app;
    VecDot(aux, p, &app);

    PetscScalar alpha = rr / app;
    VecAXPY(x, alpha, p);  // x = x + alpha * p
    VecAXPY(r, -alpha, aux);  // r = r - alpha * A*p

    PetscScalar old_rr = rr;
    VecDot(r, r, &rr);
    PetscScalar beta = rr / old_rr;
    VecAYPX(p, beta, r);  // p = r + beta * p
  }
  VecDestroy(&r); VecDestroy(&p); VecDestroy(&aux);
}
```

and supernodal Cholesky decomposition CHOLMOD (`http://www.cise.ufl.edu/research/sparse/cholmod/`). These solvers will be used in the Distributed Block Direct Solver presented in the next two chapters.

6.3.2 Optimisations

Matrix storage

In Eigen the default sparse matrices storage format is Compressed Sparse Column (CSC). It is easy to switch to Compressed Sparse Row (CSR) format using templates,

Algorithm 6.4 Conjugate Gradients implementation using Eigen library

```
void ConjugateGradients(size_t max_iter, const MatrixType& A,
                        const VectorType& b, VectorType* x) {
  VectorType r = b, p = r, aux(b.rows());
  x->setZero(b.rows());

  ScalarType rr = r.dot(r);
  for (size_t iter = 0; iter < max_iter; ++iter) {
    aux = A * p;

    ScalarType alpha = rr / p.dot(aux);
    *x += alpha * p;
    r -= alpha * aux;

    ScalarType old_rr = rr;
    rr = r.dot(r);
    ScalarType beta = rr / old_rr;
    p = r + beta * p;
  }
}
```

however the manual states that the Eigen library was designed with CSC storage in mind, and most optimisations were done for that format. This includes automatic data alignment to benefit from CPU vector instructions.

In contrast, CSR is the base format for CUSPARSE library, and there is no direct support for CSC, except for storage format conversions. In addition, CUSPARSE supports sparse matrix-vector multiplication in Block Compressed Sparse Row (BSR) and Hybrid (Ellpack-Itpack + Coordinate) formats. More information on the matrix storage formats can be found in the CUSPARSE documentation: `http://docs.nvidia.com/cuda/cusparse/#matrix-formats`.

In the case of symmetric matrices, the CSR and CSC representations are identical. Since the Conjugate Gradients method is only applicable to SPD matrices, it was decided to use CSC format on the CPU and CSR on the GPU. This way no format conversion is ever required.

Storing intermediate results

In Algorithm 6.1, it can be observed that vector $\mathbf{Ap}^{(r)}$ is used two times (lines 3 and 5). To avoid computing the sparse matrix-vector multiplication twice, the result can be stored in an auxiliary vector.

Similarly, the dot product $(\mathbf{r}^{(r)}, \mathbf{r}^{(r)})$ is used twice (lines 3 and 6). Furthermore, the dot product $(\mathbf{r}^{(r+1)}, \mathbf{r}^{(r+1)})$ in line 6 becomes the $(\mathbf{r}^{(r)}, \mathbf{r}^{(r)})$ value in the next iteration, and can be reused. In consequence, only two dot products have to be computed in each iteration, instead of four.

Fusing vector scaling and AXPY operation

The direction vector update (Algorithm 6.1, line 7) requires two BLAS calls: scaling vector $\mathbf{p}$ by scalar β, and then the AXPY operation with vector $\mathbf{r}$ and $\alpha = 1$. This implementation might have an adverse impact on performance on the CPU (the cost of running the second loop and unnecessary multiplication by one) and on the GPU (the cost of additional kernel dispatch).

To alleviate this problem the PETSc library offers the AYPX function, and Eigen does the necessary optimisations automatically using C++ templates. On the GPU, the CUDA kernel offering AYPX functionality had to be implemented manually.

Sparse matrix-vector multiplication on the GPU

In order to reduce memory requirements, only half of the symmetric matrix can be stored. This memory footprint reduction may be especially important on the GPU platforms, since they typically have smaller memories. Both Eigen and CUSPARSE libraries support SpMV on matrices stored in this way.

However, the CUSPARSE implementation of SpMV for symmetric matrices is significantly slower than when all the non-zero elements are explicitly stored (Table 6.1). The greatest differences in performance are observed for sparser large matrices — the general SpMV is up to 46 times faster in single precision, and up to 32 times faster in double precision.

Table 6.1: Symmetric sparse matrix-vector multiplication performance (memory throughput in GB/s). Storing all non-zero elements explicitly leads to 5–46 times faster execution. In general, the differences get greater as the matrices become larger and sparser.

		Single precision throughput			Double precision throughput		
Size	Non-zeros	General	Symm.	Speed-up	General	Symm.	Speed-up
65,536	326,656	34.4	1.70	20.2×	48.7	2.56	19.0×
52,804	10,614,210	89.2	8.06	11.1×	96.4	11.87	8.1×
262,144	1,308,672	55.6	1.76	31.6×	75.0	2.67	28.1×
220,542	10,768,436	84.8	10.12	8.4×	79.9	14.56	5.5×
1,048,576	5,238,784	66.0	1.80	36.6×	87.3	2.74	31.9×
1,465,137	21,005,389	90.4	1.98	45.6×	82.2	2.99	27.5×
1,437,960	63,156,690	90.3	5.83	15.5×	87.3	8.51	10.3×

Furthermore, the CUSPARSE implementation of SpMV for symmetric matrices uses atomic add instructions. In consequence, the order in which row elements are added is non-deterministic. Since addition of floating-point numbers is not associative, the symmetric SpMV may return non-identical results for the same input.

The HPSL library is designed to support solvers for non-symmetric problems. In consequence, by default all non-zero matrix elements are stored explicitly.

Diagonal Preconditioner

The efficient Diagonal Preconditioner implementation pre-computes all the inverted diagonal elements once and performs coefficient-wise vector multiplication in the solution phase. The CSR format does not allow for straightforward diagonal elements extraction. For each row, only the first and the last non-zero element index is known — the value on the diagonal is somewhere in between.

The Eigen library provides a `diagonal()` method in the `SparseMatrix` class, which performs the extraction. On the GPU, an appropriate CUDA kernel had to be implemented manually — each thread finds the diagonal element in the corresponding row using a linear search (Algorithm 6.5).

Algorithm 6.5 The CUDA kernel extracting and inverting diagonal elements of a matrix in CSR format (used in Diagonal and Polynomial Preconditioners)

```
template <typename T>
__global__ void InvDiagonal(int n, const T* values, const int* row_ptr,
                            const int* col_idx, T* diag) {
  const int row = threadIdx.x +
      blockDim.x * (blockIdx.x + blockIdx.y * gridDim.x);
  if (row >= n) return;

  const int start = row_ptr[row];
  const int end = row_ptr[row + 1];
  int idx = start;
  for (; idx < end; ++idx) {
    if (row == col_idx[idx]) break;  // Diagonal element is found
  }
  diag[row] = (idx < end ? T(1) / values[idx] : 0);
}
```

Polynomial Preconditioner

In the Polynomial Preconditioner, matrix $\mathbf{G}$ has to be computed during the construction step. In the more complex implementation $\mathbf{G} = \mathbf{I} - \mathbf{N}^{-1}\mathbf{A}$, where $\mathbf{N} = diag(\mathbf{A})$. In the solution phase Horner's scheme is performed (SpMV and vector add operations). All these operations are easy to implement in the Eigen library.

On the GPU, the construction step is not trivial. First, the inverse diagonal vector has to be computed (Algorithm 6.5). The matrix $\mathbf{G}$ has zeros on the diagonal, but otherwise it follows the same sparsity pattern as matrix $\mathbf{A}$. In consequence, if zeros on the diagonal are stored explicitly, then the column index and row pointer vectors of matrix $\mathbf{A}$ can be reused — only the values vector has to be computed.

The CUDA kernel computing values of the matrix $\mathbf{G}$ is presented in Algorithm 6.6. Since diagonal elements are treated differently, they need to be identified and set to zero; other elements have to be negated and then multiplied by a corresponding element from the inverse diagonal vector. The kernel uses one thread per non-zero element and binary search to identify the row in which the processed element lies.

Algorithm 6.6 The CUDA kernel constructing the Polynomial Preconditioner

```
template <typename T>
__global__ void Polynomial(int nnz, int n, const T* values,
                           const int* row_ptr, const int* col_idx,
                           const T* inv_diag, T* polynomial) {
  const int idx = threadIdx.x +
      blockDim.x * (blockIdx.x + blockIdx.y * gridDim.x);
  if (idx >= nnz) return;

  int left = 0, right = n;
  while (left + 1 < right) {  // Identify the row (binary search)
    const int middle = (left + right) >> 1;
    if (idx < row_ptr[middle]) {
      right = middle;
    } else {
      left = middle;
    }
  }
  const int row = left;
  const int col = col_idx[idx];
  polynomial[idx] = (row != col ? -inv_diag[row] * values[idx] : 0);
}
```

Incomplete Cholesky and LU Preconditioners

In the construction step of the preconditioners based on incomplete factorisation the triangular factors have to be computed. Then, in the solution step two triangular systems have to be computed. All the operations are supported by Eigen and CUSPARSE libraries. The latter requires an additional step to analyse the sparsity pattern of matrix $\mathbf{A}$ to identify parallelisation opportunities.

The only limitation is that the Incomplete Cholesky factorisation in CUSPARSE requires that only one triangular factor of the symmetric matrix is stored. Since this would significantly degrade the performance of the SpMV operation, two copies of matrix $\mathbf{A}$ are created in the GPU memory: one where all non-zero elements are stored explicitly, and the other storing only the lower triangle of $\mathbf{A}$. The latter is then overwritten by the $\mathbf{L}$ factor of the Incomplete Cholesky decomposition.

6.4 Numerical Experiments

The experiments were conducted on a machine equipped with an Intel Core i7-980x CPU (cf. Table B.1), running the Ubuntu 12.04 Server (64-bit) operating system. The main memory consisted of six PC3-8500 DIMMs working in triple channel mode, i.e. the theoretical maximum bandwidth was 25.6 GB/s. The CPU code was compiled with `gcc` version 4.6.3, and the GPU code was compiled with `nvcc` release 5.5 and CUDA Toolkit 5.5. The experiments were carried out on a GeForce GTX 480 graphics card with the theoretical maximum bandwidth of 177.4 GB/s (cf. Table B.2).

Unless specified otherwise, all experiments were run using double precision floating-point arithmetic, and repeated ten times. The results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5%.

6.4.1 Matrices

The starting point for all experiments was the matrix obtained with the Finite Difference Method applied to 2D Poisson's Equation on a uniform grid. All the other sparse matrices come from the University of Florida Sparse Matrix Collection (Davis and Hu, 2011) and can be downloaded from `http://www.cise.ufl.edu/research/sparse/matrices/`. Table 6.2 summarises the matrices used in this section.

(A) Original mesh (B) Distribution of non-zero elements

Figure 6.1: The ill-conditioned linear system

The last matrix represents a discretisation of the pressure correction equation, which appears in a typical CFD calculation. The pressure correction is based on pressure-velocity coupling given by the Navier-Stokes Equations and mass continuity. The

Table 6.2: Symmetric and positive-definite matrices used in the PCG experiments.

Name	Size (n)	Non-zeros (nnz)	Comment
small (5)	65,536	326,656	2D Poisson's Equation
small (200)	52,804	10,614,210	`crankseg_1`
medium (5)	262,144	1,308,672	2D Poisson's Equation
medium (7)	525,825	3,674,625	`parabolic_fem`
medium (50)	220,542	9,895,422	`hood`
large (5)	1,048,576	5,238,784	2D Poisson's Equation
large (15)	1,465,137	21,005,389	`StocF-1465`
large (45)	1,437,960	63,156,690	`Geo_1438`
ill-conditioned	768	4,864	Condition number $\kappa \sim 10^{12}$

equation is used to find a pressure field that will induce a flow satisfying mass continuity (Ferziger and Perić, 2002, Ch. 7). This particular matrix is a part of a volume of fluid simulation for gas-liquid flow inside a straight pipe section and discretised using a so called O-mesh arrangement (Figure 6.1 A), i.e. square section in the middle and four sections based on arcs and adjacent to square sides. This mesh arrangement is used to avoid the singularity that occurs in an approach based on cylindrical coordinates. The resulting linear system (Figure 6.1 B) is particularly interesting since even for a small mesh ($n = 768$), it has a very high condition number ($\kappa \sim 10^{12}$) which makes it difficult to solve with iterative methods.

6.4.2 Performance Models

All steps in the Conjugate Gradients solver require time proportional to the input data size, and have a low computation-to-communication ratio, therefore the whole CG solver is memory-bound. To enable meaningful analysis, the following performance models based on memory throughput were used (here n denotes the matrix size, nnz — number of non-zero elements, R — the number of CG iterations, S_{val}

— the size of scalars in bytes, S_{idx} — the size of indices in bytes):

$$S_{spmv} = S_{val} \cdot (nnz + 2n) + S_{idx} \cdot (nnz + n + 1) \tag{6.10}$$

$$S_{axpy} = S_{val} \cdot (2n + 1) \tag{6.11}$$

$$S_{dot} = S_{val} \cdot (2n + 1) \tag{6.12}$$

$$S_{cg} = R \cdot S_{spmv} + 3R \cdot S_{axpy} + (2R + 1) \cdot S_{dot} \tag{6.13}$$

6.4.3 Relative Performance of the Selected Libraries

In the first experiment, the classic Conjugate Gradients implementations included in the High-performance Parallel Solvers Library (HPSL) are compared against the existing well-established libraries. The execution times were normalised against the HPSL library, i.e. values lower than one indicate that the solver was faster than the HPSL implementation.

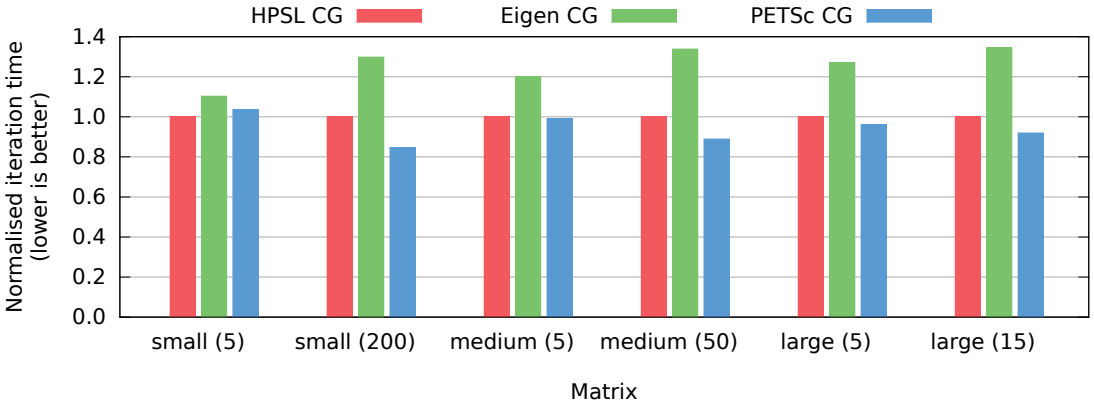


Figure 6.2: Performance comparison of the selected libraries offering Conjugate Gradients implementation on the CPU. For matrices with more non-zero elements per row, the PETSc library is slightly faster than HPSL due to more efficient SpMV implementation.

The PETSc library is slightly faster than HPSL (Figure 6.2). The difference does not depend on the linear system size, but is more significant for denser matrices and negligible for the sparser systems. The better performance of PETSc can be explained by a faster SpMV implementation, however HPSL is only up to 18% slower. This is the overhead for a higher level of abstraction, easier programming, integration, and extensibility of HPSL. The black-box Conjugate Gradients solver available in the Eigen library is consistently slower (20–40%) than HPSL implementation.

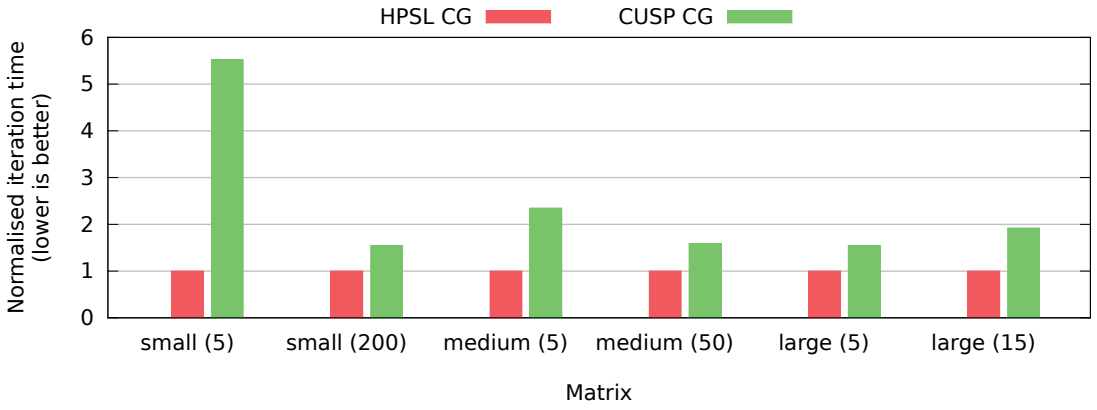


Figure 6.3: Performance comparison of the selected libraries offering Conjugate Gradients implementation on the GPU. HPSL is significantly faster than the CUSP library, especially for matrices with fewer non-zero elements per row.

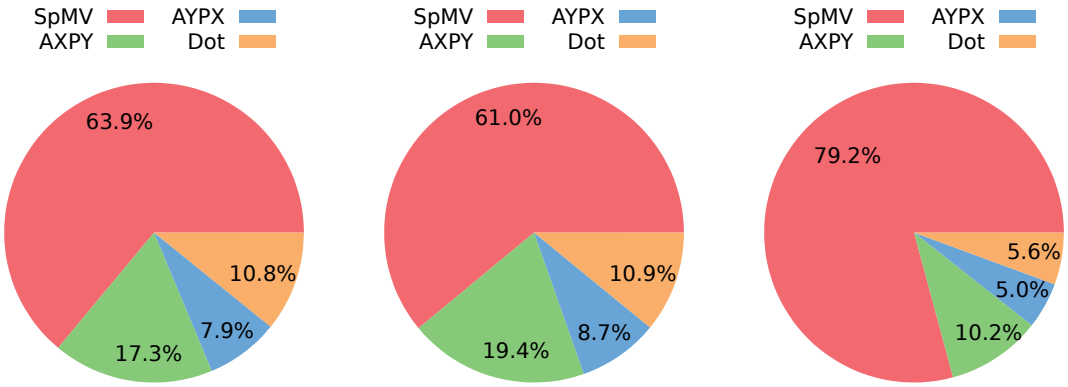
On the GPU, HPSL was compared against the black-box Conjugate Gradients implementation available in the CUSP library. Also in this case the HPSL library is superior — CUSP is at least 50% slower (Figure 6.3). The difference becomes more prominent for linear systems with fewer non-zero elements per row: for the smallest matrix considered in this experiment HPSL is over five times faster.

6.4.4 The Performance of Conjugate Gradients Operations

The goal of the next experiment was to determine the performance of each operation comprising the Conjugate Gradients and to identify performance-limiting factors.

As expected, sparse matrix-vector multiplication is the most time-consuming step in the CG solver (Figure 6.4). Unlike all the other operations, SpMV depends on the number of non-zero elements. In consequence, SpMV takes an even greater fraction of execution time for a matrix with more non-zero elements per row (Figure 6.4c). There are no significant differences in the execution time structure between the CPU (Figure 6.4a) and the GPU (Figure 6.4b).

The next step was to assess the effective memory throughput of each Conjugate Gradients step. The throughputs were calculated using performance models presented in Subsection 6.4.2. Sparse matrix-vector multiplication is the slowest operation on



(A) CPU, large (5) (B) GPU, large (5) (C) GPU, large (15)

FIGURE 6.4: Conjugate Gradients execution time breakdown. There are no significant differences between the CPU (A) and the GPU (B). SpMV takes greater fraction of execution time for a denser matrix (C) since it is the only operation depending on the number of non-zero elements (all the other operations depend on the matrix size only).

both devices, due to the least cache-friendly memory access pattern (matrix elements are accessed only once, but vector elements are possibly accessed frequently in a non-sequential order). Since SpMV is the dominant operation (Figure 6.4), its performance has a direct impact on the overall Conjugate Gradients method speed.

The peak theoretical memory bandwidth on the CPU (25.6 GB/s) is calculated based on the assumption that the load on main memory is equally distributed among all three memory channels. However, this cannot be controlled by a programmer and in many cases the effective memory bandwidth can be as low as that of a single channel, i.e. 8.533 GB/s. This would explain the relatively low throughput of sparse matrix-vector multiplication oscillating between 6.5 and 8.5 GB/s (Figure 6.5).

All the other Conjugate Gradients steps are performed at much higher rates, even exceeding the theoretical maximum main memory bandwidth, due to high cache hit ratio. The significant impact of cache on overall performance is confirmed by the fact that the performance is decreasing with increasing linear system size. Indeed, the lowest throughput is observed for the large linear systems, for which only one vector can fit in the L3 cache at a time.

On the CPU, the Conjugate Gradients solver achieves the best performance for small linear systems with fewer non-zero elements per row (up to 13.5 out of 25.6 GB/s, or

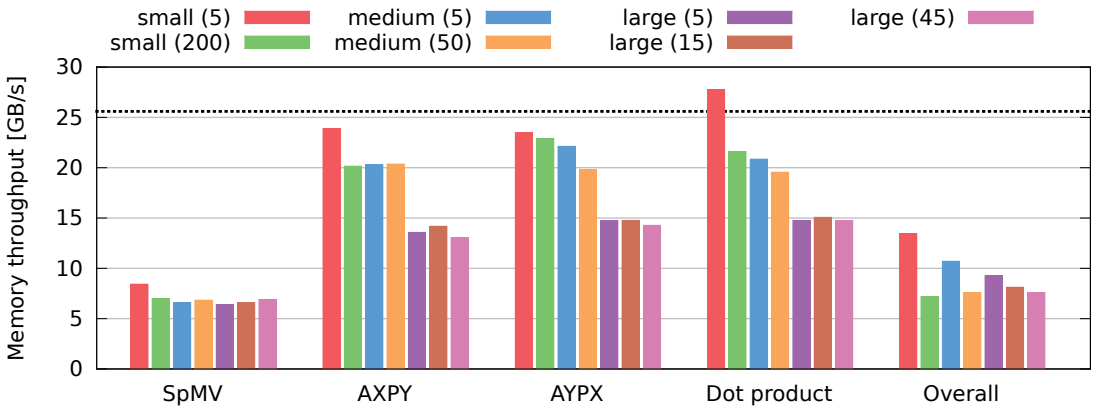


Figure 6.5: Memory throughput observed for CG steps on Intel i7-980x. The dotted line marks the theoretical peak performance (25.6 GB/s). The high throughput observed for AXPY, AYPX, and dot product on smaller matrices is due to higher cache hit ratio.

53% of the theoretical peak). This is connected with the decreasing cache hit ratio as the input data gets bigger. The throughput observed for SpMV does not degrade with the increasing number of non-zero elements. However, it is the slowest operation and its fraction of execution time for denser matrices is increasing, leading to an overall decrease in performance (which stays above 30% of the theoretical peak).

On the GPU, the throughput observed for sparse matrix-vector multiplication oscillates at roughly half of the theoretical peak performance (Figure 6.6). The only exception is system small (200) where the number of non-zero elements per row is high, and the vectors are small enough to fit in global memory cache. This allows SpMV to run 25% faster than for the other linear systems.

The dot product, and AXPY and AYPX operations depend only on the vector length, which explains why there are no differences between linear systems of a similar size. Unlike the CPU, on the GPU observed throughputs increase with the increasing problem size due to better device utilisation, and for large matrices these operations execute at 90% of the theoretical peak performance. It is not possible to exceed the theoretical peak performance on the GPU, since the global memory cache is too small (768 KB) to store the necessary data between Conjugate Gradients steps. The throughput observed for the AYPX operation is high regardless of the problem size, which confirms that the proposed manual implementation is efficient.

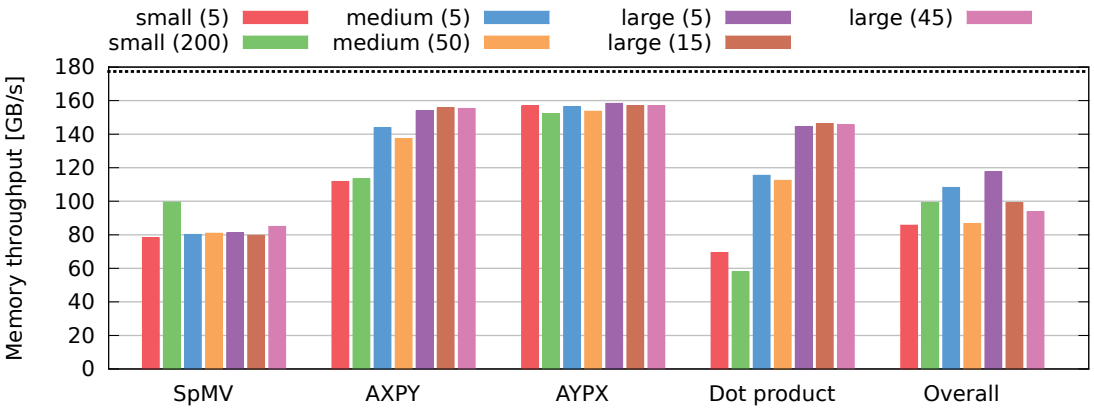


Figure 6.6: Memory throughput observed for CG steps on GeForce GTX 480. The dotted line marks the theoretical peak (177.4 GB/s). The lower performance observed for AXPY and dot product on smaller matrices is due to low device utilisation. The manual implementation of AYPX operation achieves better occupancy, i.e. utilises the GPU better.

On the GPU, the Conjugate Gradients solver achieves the best performance for large linear systems with few non-zero elements per row (up to 120 out of 177.4 GB/s, or 68% of the theoretical peak). Even though the throughput observed for SpMV does not degrade with the increasing number of non-zero elements, it is still the slowest operation and its fraction in overall execution time is increasing, leading to an overall decrease in performance (but which always stays above 50% of the theoretical peak).

6.4.5 Preconditioner Construction and Solution Times

Typically, using more complex preconditioners results in more significant reduction of the number of iterations required to reach the desired error tolerance. However, this is also connected with the more significant preconditioner construction cost and the cost of applying the preconditioner in each iteration. The remaining three subsections aim to investigate this trade-off for four different preconditioners: Diagonal, Polynomial, Incomplete LU, and Incomplete Cholesky.

The characteristics of various preconditioners may lead to a significantly different performance depending on the computational platform. To make the results meaningful, the preconditioner construction and application times were normalised

against the cost of an iteration of the Conjugate Gradients method without a preconditioner, i.e. an additional cost equal to one means that the iteration of the preconditioned CG requires twice the time of the classic CG method (Figure 6.7).

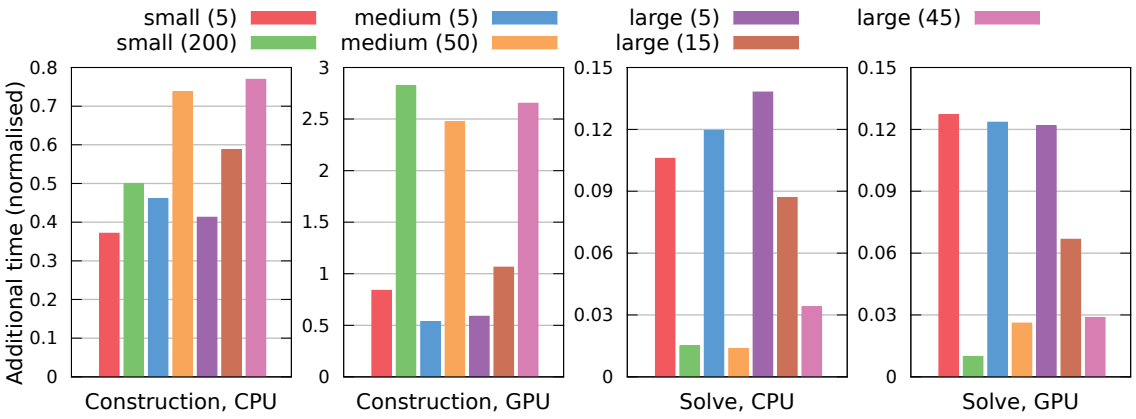
The Diagonal Preconditioner is the simplest and the fastest one — construction and solution costs are almost negligible (Figure 6.7 A). The latter cost is comparable on both devices. It is lower for matrices with more non-zero elements per row, due to higher fraction of time spent on SpMV operation, effectively hiding the cost of applying the preconditioning matrix. Nevertheless, even for matrices with the lowest density the cost does not exceed 15% of the classic CG iteration.

The construction cost of the Diagonal Preconditioner increases with the number of non-zeros per row. This is connected with the fact, that the CSR storage format does not allow for straightforward extraction of diagonal elements. This process is even less efficient on the GPU, leading to higher normalised construction cost.

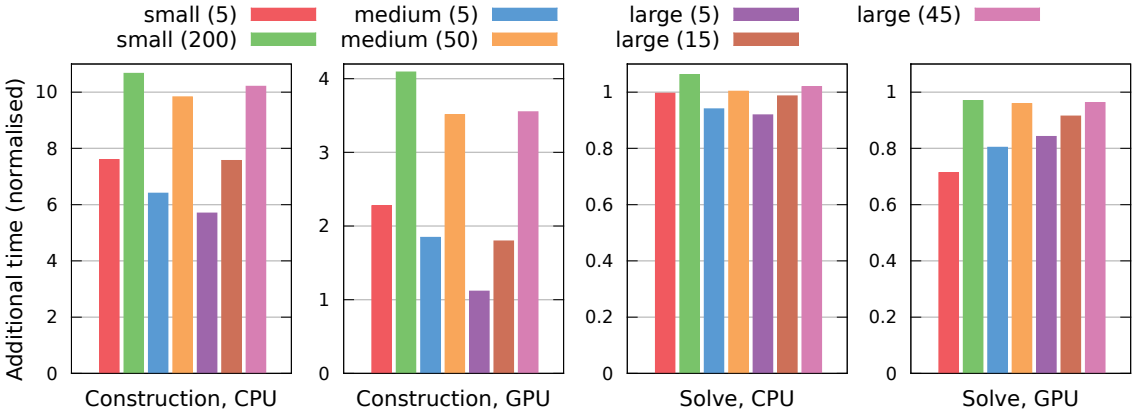
The construction of the Polynomial Preconditioner involves allocation, and scaling $\mathbf{G}$ which has sparsity pattern similar to the original linear system, hence the cost is proportional to the matrix size and the number of non-zero elements. In consequence, the normalised cost increases with the density of the linear system, and amounts to a few iterations of the classic Conjugate Gradients solver (Figure 6.7 B). The lower relative cost on the GPU is connected with faster memory allocation.

Unlike the construction, the solution step of the Polynomial Preconditioner depends on the degree of the preconditioner. The presented results are for the Polynomial Preconditioner of the first degree and the solution time increases in a linear fashion with the degree. The dominant operation is the SpMV on the matrix that has an identical non-zero pattern as the original system. Therefore, it was expected that the cost of preconditioning would be close to the classic CG iteration. Small differences (cost increasing with the matrix density) are connected with the fraction of execution time spent on the SpMV operation.

In the case of incomplete factorisation preconditioners, the time complexity of the construction and solution steps are an order of magnitude higher leading to significantly greater normalised costs (Figures 6.7 C and 6.7 D). The preconditioner construction cost greatly depends on the pattern of non-zero elements — for 2D Poisson matrices (up to five non-zeros per row) the factorisation time is negligible,



(A) Diagonal Preconditioner

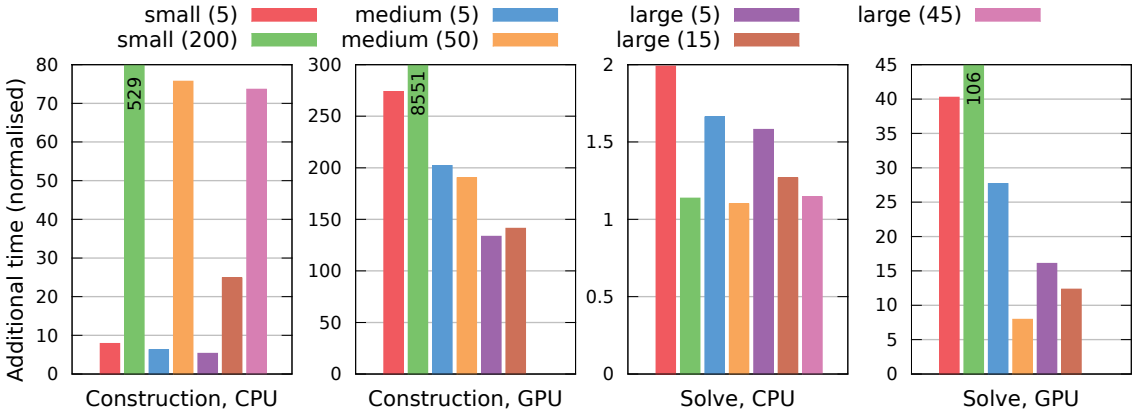


(B) Polynomial Preconditioner (first degree)

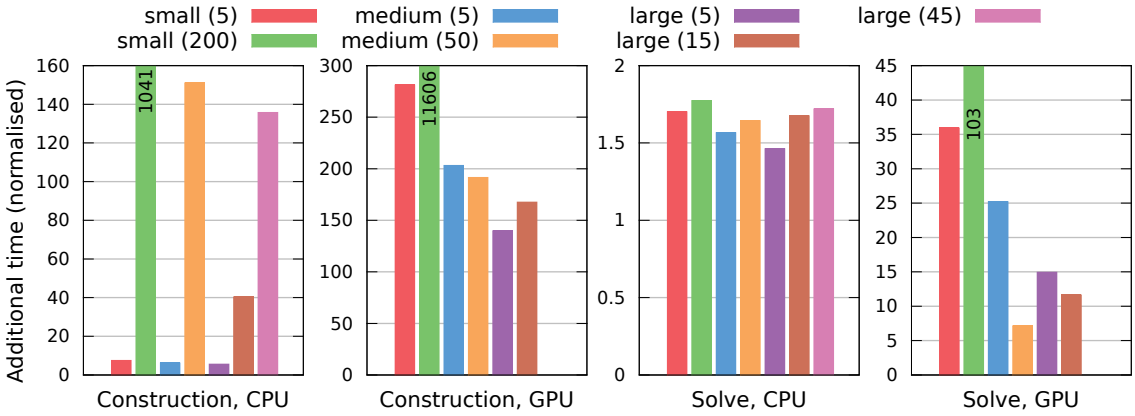
Figure 6.7: Construction and solution time of various preconditioners on the CPU (Intel i7-980x) and the GPU (GTX 480), normalised against a single CG iteration The Diagonal Preconditioner is the fastest. The construction on the GPU is longer for denser matrices due to non-trivial diagonal elements extraction. The Polynomial Preconditioner is based on SpMV operation, therefore the solution step requires roughly the time needed for one CG iteration. The construction is also proportional to the number of non-zero elements. Relatively larger cost on the CPU is connected with slower memory allocation.

Continued on the next page

Continued from the previous page



(C) Incomplete Cholesky Preconditioner



(D) Incomplete LU Preconditioner

The construction phase of incomplete factorisation preconditioners has $\mathcal{O}(nnz^2)$ complexity and can be prohibitively slow for denser matrices. The relative performance on the CPU is as expected: Cholesky factorisation (C) is two times faster than LU (D), and the solution step is relatively fast ($\mathcal{O}(n+nnz)$ complexity). Due to limited parallelisation opportunities, the performance of both preconditioners on the GPU is significantly lower. Furthermore, the construction phase requires a significant amount of auxiliary storage — there was not enough memory to factorise the largest matrix on the GPU.

however for the most dense matrix (small, 200) decomposition is prohibitively slow and renders incomplete factorisations impractical as preconditioners in this case. The normalised cost gets even worse on the GPU — factorisation requires the time of hundreds, or even thousands of CG iterations.

Typically, the Cholesky method is considered to be twice as fast as the more general LU factorisation. This trend can be observed on the CPU, especially for denser matrices. However, on the GPU differences are much smaller. In fact, the Incomplete Cholesky factorisation is faster only for linear systems with a high number (> 50) of non-zero elements per row. Even then, the Incomplete LU factorisation is only up to 36% slower. The smaller differences are a consequence of low device utilisation (number of active threads) due to limited parallelisation opportunities.

The solution step for incomplete factorisation preconditioners consists of one forward and one backward substitution. The time complexity of these operations depends on the number of non-zero elements in the triangular factors. In the case of the Incomplete LU factorisation on the CPU, the normalised cost does not depend on the linear system and oscillates around 160% of the CG iteration time (Figure 6.7 D).

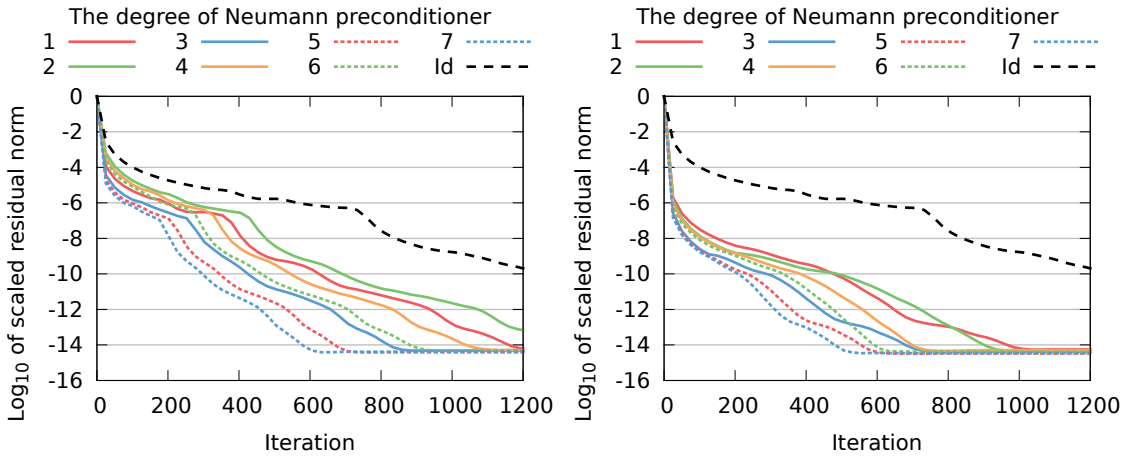
In contrast, the normalised cost of the solution step in the Incomplete Cholesky factorisation on the CPU varies between one and two CG iterations, with a decreasing trend with respect to the increasing matrix density (Figure 6.7 C). It is important to note that the upper Cholesky factor is the transposition of the lower factor. The implementation proposed in this chapter takes advantage of this fact and stores only one Cholesky factor, which is effectively cached during the first substitution step. In consequence, in the second substitution the data is read from cache, and the speed-up is bigger when the number of non-zero elements is greater.

The normalised cost of the solution step is significantly higher on the GPU, again due to the limited parallelisation opportunities. In contrast to the CPU case, there are no significant differences in the solution time between Cholesky and LU factorisations. Even though only one Cholesky factor is stored, the global memory cache on the GPU (768 KB) is too small to store all the non-zero elements. In fact, the solution step in the Incomplete LU Preconditioner case is slightly faster — the CUSPARSE library can take advantage of the fact that one of the LU factors has a unit diagonal (all elements are equal to one) and it does not need to be loaded from global memory.

6.4.6 The Optimal Polynomial Preconditioner

The cost of applying a Polynomial Preconditioner is proportional to the number of non-zero elements and the degree of the preconditioner. The solution step of the Polynomial Preconditioner (of degree K) requires time corresponding to K iterations of the classic CG method (Figure 6.7 B). Therefore, it only makes sense to use this preconditioner if the number of iterations is reduced at least by a factor of $(K+1)$.

To compare two versions of the Polynomial Preconditioner considered in this study and their ability to improve the original Conjugate Gradients method, scaled residual norms were plotted against the number of iterations and computation time. The experiments were carried out on the medium (7) matrix (`parabolic_fem`).



(A) $\mathbf{G} = \mathbf{I} - \omega\mathbf{A}$ (Liang, 2005)

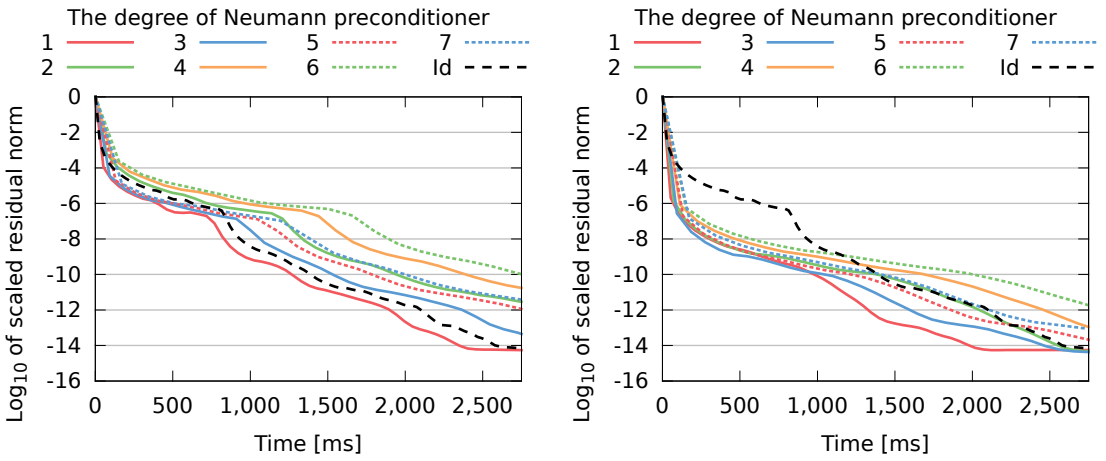
(B) $\mathbf{G} = \mathbf{I} - \mathbf{N}^{-1}\mathbf{A}$ (Becker, 2006)

FIGURE 6.8: Convergence rate of the PCG with various Polynomial Preconditioners for the `parabolic_fem` matrix. Id denotes the classic CG method. The most significant reduction in the number of iterations is observed for high-degree preconditioners, however there is no clear correlation between the convergence rate and polynomial degree.

Applying the Polynomial Preconditioner can significantly improve the rate of convergence (Figure 6.8). The lowest number of iterations is required for high-degree preconditioners, however there is no correlation between the polynomial degree and convergence rate. The best results were observed for polynomials of degree 5 and 7.

There are no significant differences in convergence rate trends between both Polynomial Preconditioners, however for the same polynomial degree the simpler pre-

conditioner (Figure 6.8A) converges slower than the more complex implementation (Figure 6.8B). The latter preconditioner requires 5–15% fewer iterations to converge to a solution of the same quality.



(A) $\mathbf{G} = \mathbf{I} - \omega\mathbf{A}$ (Liang, 2005)

(B) $\mathbf{G} = \mathbf{I} - \mathbf{N}^{-1}\mathbf{A}$ (Becker, 2006)

FIGURE 6.9: PCG solution quality in time with various Polynomial Preconditioners for the `parabolic_fem` matrix on the Intel i7-980x. Id denotes the classic CG method. Only the low degree Polynomial Preconditioners offer good time-effectiveness. The performance of the more complex preconditioner (B) is better, regardless of the polynomial degree.

Since the cost of both versions of the Polynomial Preconditioner is comparable, it was expected that trends in solution quality over time will be similar. However, the preconditioning overhead may be significant in comparison to the time saved on fewer iterations. Both preconditioners are more time-effective for lower degree polynomials (Figure 6.9).

In the case of the simpler Polynomial Preconditioner, the performance improvement can only be observed for a polynomial of the first degree (Figure 6.9A). Even then, the PCG solver is only 10% faster than the classic Conjugate Gradients method.

The more complex Polynomial Preconditioner is more time-effective than the simpler variant, regardless of the polynomial degree (Figure 6.9B). The PCG solver is faster than the classic CG method with polynomials of degree 1 (over 25% speed-up) and 3 (roughly 7.5% speed-up). In consequence, only the more complex Polynomial Preconditioner of the first degree will be used in further experiments.

6.4.7 Preconditioner Comparison

The goal of the last set of experiments was to assess the performance of the Preconditioned Conjugate Gradients method with different preconditioners in solving various linear systems.

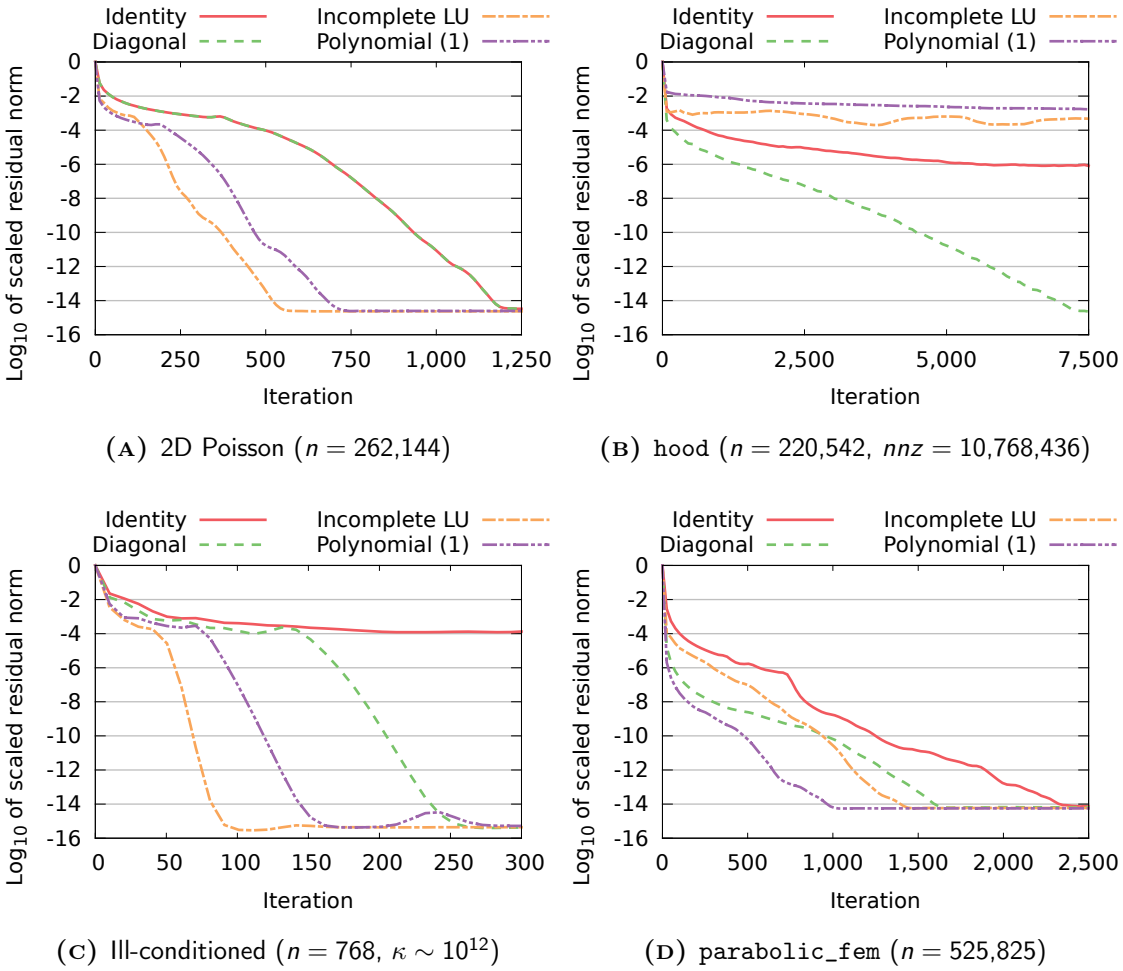
In almost all cases, using a preconditioner leads to a reduced number of iterations being required to reach the desired error tolerance (Figure 6.10). However, the level of reduction strongly depends on the properties of a particular matrix. Since all matrices considered in this study are symmetric, both incomplete factorisations lead to the same solution, therefore plots for Incomplete Cholesky were omitted.

The Diagonal Preconditioner does not have any effect in the case of the 2D Poisson matrix (Figure 6.10 A). However, the other two preconditioners allow us to reduce the number of iterations required to converge to double precision machine error from 1,250 down to 700 (Polynomial), or even 500 (Incomplete LU).

In contrast, the Diagonal Preconditioner offers the best convergence rate for the `hood` matrix (Figure 6.10 B). In this case, the magnitude of the diagonal elements is lower than the magnitude of the sum of the non-diagonal elements in a row (the matrix is not diagonally dominant). Application of the Diagonal Preconditioner results in a reduction of the condition number by a factor of roughly one million, and convergence in roughly a thousand times fewer iterations. All the other preconditioners actually deteriorate the convergence rate and are impractical in this case.

For a small, ill-conditioned, and diagonally dominant linear system, all preconditioners offer a significant improvement in the rate of convergence, especially the preconditioners based on incomplete factorisations (Figure 6.10 C). Using the Incomplete LU Preconditioner allows us to reduce the number of iterations required to converge to machine precision error from 1,300 down to less than a hundred.

Similarly, in the case of the last linear system all preconditioners allow us to improve the convergence rate, however not that significantly (Figure 6.10 D). This is connected with the relatively low condition number ($\kappa \approx 13$) of the `parabolic_fem` matrix. The best improvement was observed for the Polynomial Preconditioner — the number of required iterations was reduced from 2,300 down to a thousand.



(A) 2D Poisson ($n = 262{,}144$)

(B) `hood` ($n = 220{,}542$, $nnz = 10{,}768{,}436$)

(C) Ill-conditioned ($n = 768$, $\kappa \sim 10^{12}$)

(D) `parabolic_fem` ($n = 525{,}825$)

Figure 6.10: PCG convergence rate comparison with various preconditioners and different linear systems. Typically, more complex preconditioners allow for greater reduction in the number of iterations (A) and (C). For the `hood` matrix only the Diagonal Preconditioner allows for an improvement in convergence rate (B), due to unfavourable matrix properties (high condition number, not diagonally dominant). The `parabolic_fem` matrix has low condition number and improvement from incomplete factorisation preconditioners is limited (D) — in this case, the Polynomial Preconditioner offers the best reduction in the number of iterations required to converge to machine precision.

The highest reduction in the number of iterations required for the PCG method to converge does not automatically mean that a particular preconditioner is the most time-effective, since the cost of preconditioning can be significant (Subsection 6.4.5). To factor that into the preconditioner comparison, the scaled residual norms were plotted against the total execution time (Figure 6.11). The preconditioner construction time was not included, since often the same system is solved with multiple right-hand sides (the preconditioner has to be constructed only once).

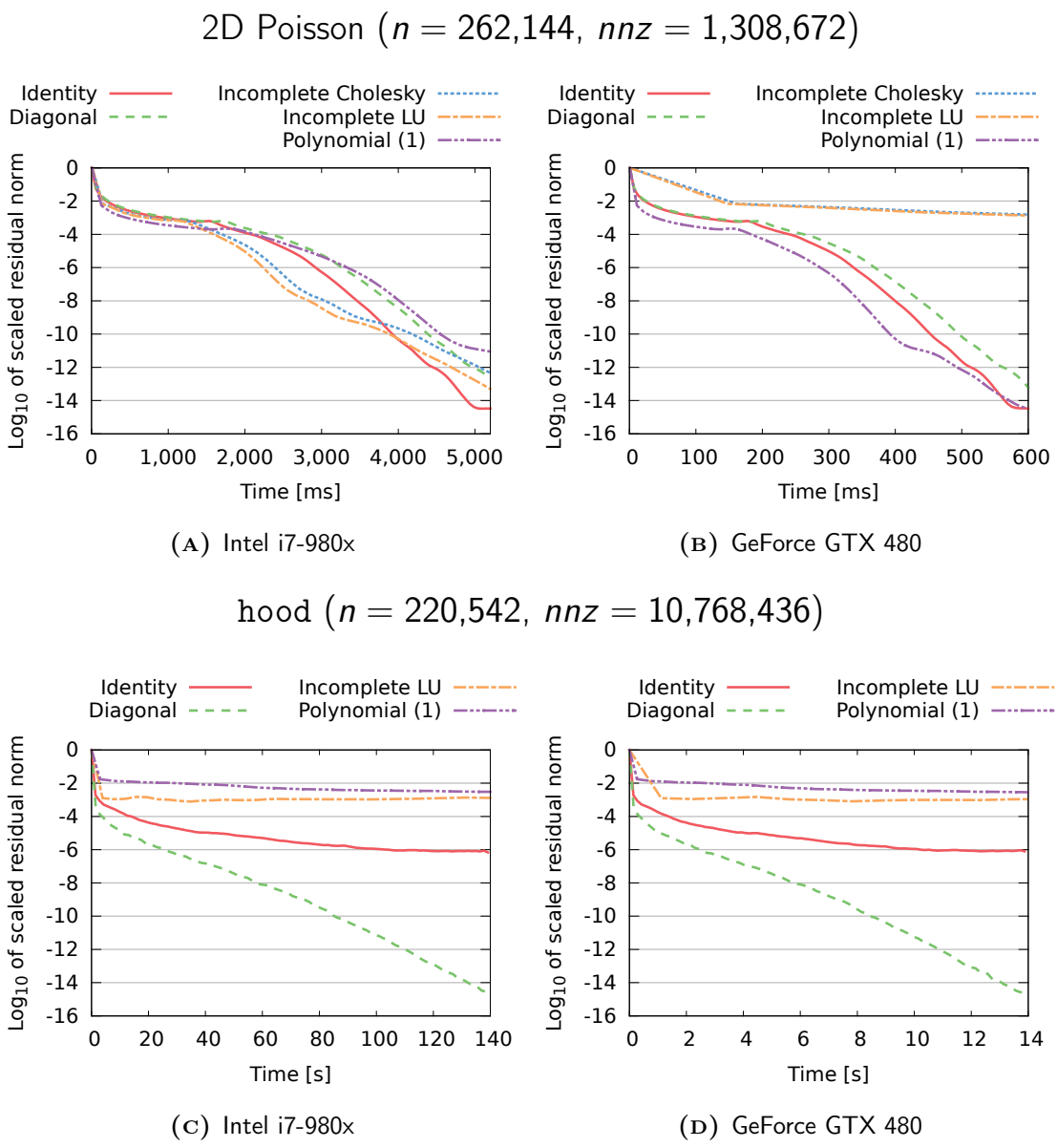
In the case of the 2D Poisson matrix on the CPU, the benefits from an improved convergence rate are balanced by higher preconditioning costs. In consequence, there are no significant differences between various preconditioners (Figure 6.11 A). After 3 seconds the best solution is offered by PCG with Incomplete LU factorisation, however the classic CG is the first (5 seconds) to reach machine precision.

On the GPU, the incomplete factorisations are significantly slower and require roughly 6 seconds to converge. All other methods are comparable, however PCG with the Polynomial Preconditioner is slightly faster and converges to machine precision in less than 600 ms (Figure 6.11 B).

In the case of the `hood` matrix, trends on the CPU (Figure 6.11 C) and the GPU (Figure 6.11 D) are similar — PCG with the Diagonal Preconditioner is clearly the fastest method. However, the GPU solver is roughly ten times faster to converge to machine precision.

Due to a significant reduction in the number of iterations, PCG with Incomplete LU (Cholesky) and Diagonal Preconditioners is the fastest to converge on the CPU for the small, ill-conditioned, and diagonally dominant system (Figure 6.11 E). The Polynomial Preconditioner also offers a noticeable improvement over the classic CG method, however it requires roughly double the time of the fastest solver.

In contrast, on the GPU PCG with the Polynomial Preconditioner is the most time-effective method, however it requires roughly four times more time to converge than the CPU implementation (Figure 6.11 F). The linear system with just 768 equations is too small to be efficiently processed on the GPU — CUDA thread management overheads are greater than the actual computation time. In consequence, the GPU is idle for most of the time. This is also the reason why the Polynomial Preconditioner



(A) Intel i7-980x (B) GeForce GTX 480

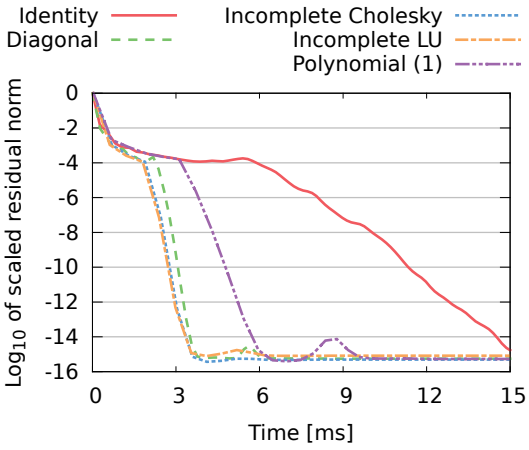
(C) Intel i7-980x (D) GeForce GTX 480

Figure 6.11: Solution quality in time of PCG with various preconditioners and different linear systems. For 2D Poisson matrix the benefits from lower number of iterations are balanced by preconditioning overheads — there are no significant differences in time-effectiveness (A) and (B). The incomplete factorisations on the GPU are significantly slower due to limited parallelisation opportunities (B). The hood matrix has unfavourable properties, which can only be improved by the Diagonal Preconditioner (C) and (D).

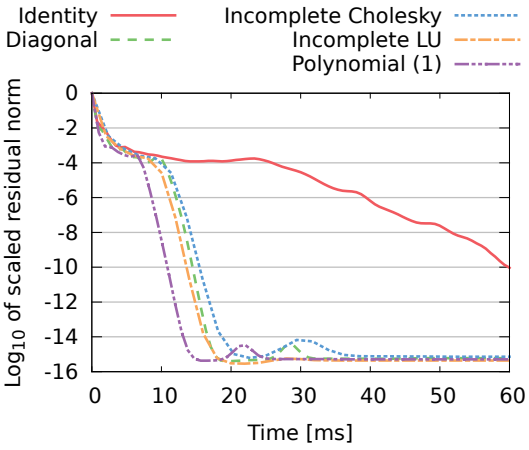
Continued on the next page

Continued from the previous page

ill-conditioned ($n = 768$, $nnz = 4{,}864$, $\kappa \sim 10^{12}$)

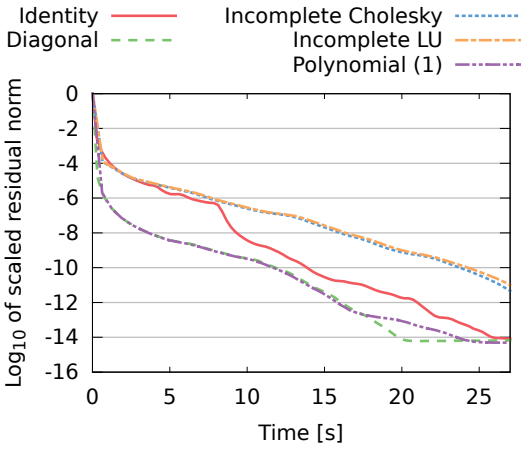


(E) Intel i7-980x

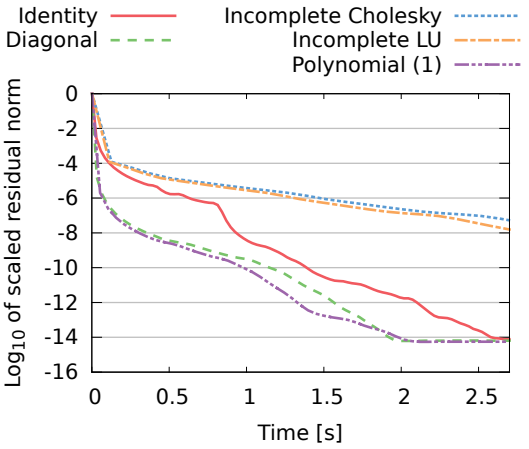


(F) GeForce GTX 480

`parabolic_fem` ($n = 525{,}825$, $nnz = 3{,}674{,}625$)



(G) Intel i7-980x



(H) GeForce GTX 480

For an ill-conditioned system, all preconditioners offer great improvement in time-effectiveness due to significant reduction in the number of iterations. The Polynomial Preconditioner is the best on the GPU due to better device utilisation (F). In case of small linear systems the solver on the GPU is several times slower, due to relatively high kernel dispatch overheads. Relatively cheap Diagonal and Polynomial Preconditioners are the most time-effective for the `parabolic_fem` matrix, due to its lower condition number.

became the fastest method — using the SpMV operation allows it to attain much better GPU utilisation level.

For the `parabolic_fem` matrix the time-effectiveness trends are similar on the CPU (Figure 6.11 G) and on the GPU (Figure 6.11 H). The PCG method with Diagonal and Polynomial Preconditioners offers the fastest convergence, however only by 20% faster than the classic CG method. Incomplete factorisation preconditioners are noticeably slower on both platforms. Similar to the `hood` matrix case, the execution time on the GPU is roughly ten times faster than on the CPU, and requires less than two seconds to converge to machine precision in double precision.

6.5 Conclusions

- This chapter investigates efficient methods to implement the Preconditioned Conjugate Gradients solver on the CPU and the GPU platforms. The preconditioners considered in this study include the Diagonal Preconditioner, the Polynomial Preconditioner, and preconditioners based on incomplete factorisations (Cholesky and LU).
- The proposed implementations are built on top of the existing libraries (Eigen, CUBLAS, CUSPARSE) and then optimised. The codes are part of the High-performance Parallel Solvers Library (HPSL), developed as a part of this PhD project, and achieve up to 53% (13.5 GB/s) of the theoretical peak performance on the CPU platform based on Intel i7-980x, and up to 68% (120 GB/s) of the theoretical peak performance on the GPU platform with GeForce GTX 480.
- The Diagonal Preconditioner is the fastest to be constructed, and its solution step increases the iteration time by only 1–15%. The construction of the Polynomial Preconditioner is slightly longer, but can still be completed in the time required for several CG iterations. The solution step in each iteration requires the same time as an iteration of the classic CG method. The relative cost of both preconditioners is similar on the CPU and on the GPU platforms.
- More complex preconditioners based on incomplete decompositions (Cholesky and LU) may require a prohibitively long time for matrix factorisation, depending on the pattern of non-zero elements. On the CPU, the construction phase

is roughly two times faster for the Cholesky method compared to LU. The solution step requires time equal to two iterations of the classic CG method.

- Due to limited parallelisation opportunities, the relative cost of incomplete factorisation preconditioners is significantly higher on the GPU platforms. The decomposition may require times of hundreds, or even thousands of iterations of the classic CG method. Then, the solution step increases the execution time of each iteration by a factor in the range 10–100, depending on the matrix sparsity pattern.

- In the case of the Polynomial Preconditioner, the most significant reduction in the number of required iterations is observed for high-degree preconditioners, however there is no clear correlation between the convergence rate and polynomial degree. In addition, using a high degree preconditioner is connected with significantly higher overhead. The experiments confirmed that the Polynomial Preconditioner of the first degree is the most time-effective.

- Typically, using more complex preconditioners should result in a more significant improvement in the convergence rate. However, this strongly depends on properties of a particular linear system. This chapter presents examples for which each of the considered preconditioners offers the best convergence rate.

- The preconditioner offering the best convergence rate is not necessarily the most time-effective due to greatly varying additional costs. The Diagonal and Polynomial Preconditioners impose relatively low overhead and can significantly reduce the condition number of the original system. Furthermore, they parallelise well and offer good performance on the GPU platforms.

- In contrast, preconditioners based on incomplete factorisations impose significant overheads. The additional cost is especially severe on the GPU platforms due to limited parallelisation opportunities. Typically, the reduction in the number of required iterations is not enough to compensate for the preconditioning overheads. Of the matrices considered in this study, the Incomplete LU (Cholesky) Preconditioner was the most time-effective in one case only — the small and ill-conditioned linear system. Even in this case, PCG with Diagonal and Polynomial Preconditioners offered comparable time-effectiveness.

- In most cases, using the Diagonal Preconditioner is the best choice. In the worst case the number of iterations stays the same and the solver runs 15% slower than the classic CG method. However, for many matrices arising from the discretisation of PDEs the convergence rate is significantly improved, especially for linear systems which are not diagonally dominant.

Chapter 7

Derivation and Properties of the Distributed Block Direct Solver

The Distributive Conjugate Gradients (DCG) method (Becker, 2006; Becker and Thompson, 2006) is a recently published PDE solution algorithm offering parallelisation of the classic Conjugate Gradients (CG) solver on distributed memory systems. DCG employs Additive Schwarz domain decomposition, which allows for almost independent concurrent solution of the subproblems. However, the DCG does not maintain some important theoretical properties of the CG method. Furthermore, the DCG convergence rate degrades when the number of subdomains is increased. Still, published experiments showed good DCG performance for several linear systems. The analysis of DCG and its properties is presented in Section 7.1.

Motivated by the DCG method, a modified algorithm is proposed in this project — the Distributed Block Direct Solver (DBDS) that is based on the Block Relaxation scheme (cf. Subsection 2.3.4). DBDS aims at improving the convergence rate, allowing for straightforward hybrid MPI-CUDA parallelism, and enabling easier analysis of the theoretical properties. The DBDS method, along with theoretical and empirical results on its rate of convergence, is presented in Section 7.2.

The DBDS method is more than just a solver — it provides a framework for linear systems solution, and can be adjusted to particular problems. The available range of extensions and their applicability is summarised in Section 7.3. Finally, the conclusions are given in Section 7.4.

7.1 Analysis of Distributive Conjugate Gradient

Distributive Conjugate Gradients (DCG) is a recently proposed parallel PDE solution method. The DCG is derived from the classic Conjugate Gradients solver (Subsection 7.1.1). The DCG convergence analysis presented in the original work (Becker, 2006) is limited and is briefly summarised in Subsection 7.1.2.

7.1.1 Derivation of the Distributive Conjugate Gradient Method

In the classic Conjugate Gradients (CG) method the quadratic form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c\,, \tag{7.1}$$

is minimised in each iteration by taking steps in *mutually conjugate* directions. If the coefficient matrix is symmetric and positive-definite (SPD), then the quadratic form defined in Equation (7.1) has exactly one minimum, namely $\mathbf{x}$ satisfying the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$. For a linear system with n equations, in exact arithmetic the CG method finds the solution in up to n iterations.

The Distributive Conjugate Gradients method employs Additive-Schwarz domain decomposition, in which P subproblems are created, effectively dividing the coefficient matrix $\mathbf{A}$, and vectors $\mathbf{x}$ and $\mathbf{b}$ into P horizontal stripes (Becker, 2006, §6.1). The decomposed problem can be formulated as

$$\begin{bmatrix} \mathbf{L}_1 & \mathbf{H}_{1,2} & \cdots & \mathbf{H}_{1,P} \\ \mathbf{H}_{2,1} & \mathbf{L}_2 & \cdots & \mathbf{H}_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{H}_{p,1} & \mathbf{H}_{p,2} & \cdots & \mathbf{L}_P \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_P \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_P \end{bmatrix} \tag{7.2}$$

Each processing element solves the local linear system defined by the coefficient matrix $\mathbf{L}_p$ — the corresponding diagonal block in the original matrix. The remaining elements in the stripe form matrix $\hat{\mathbf{H}}_p = \left[\hat{\mathbf{H}}_{p,1} \ldots \hat{\mathbf{H}}_{p,P}\right]$ and describe communication with other subdomains. Matrices $\hat{\mathbf{H}}_p$ are typically sparse, e.g. in the case of the 2D Poisson's Equation, they contain up to two non-zero elements per row.

Similar to the CG method, in each DCG iteration every processing element updates the direction vector and determines the step size for the local system

$$\mathbf{L}_p \mathbf{x}_p^{(r)} = \mathbf{b}_p - \hat{\mathbf{H}}_p \hat{\mathbf{x}}_p^{(r-1)}\,, \tag{7.3}$$

where $\hat{\mathbf{x}}_p^{(r-1)}$ is a vector of current solution values in other subdomains (Becker, 2006, p. 158). Since the right-hand side of Equation (7.3) changes after each iteration, it is no longer possible to ensure mutual conjugacy of all direction vectors. Instead, certain adjustments are made to ensure conjugacy of every two successive direction vectors (Algorithm 7.1). As in the CG method, the main cost comes from sparse matrix-vector multiplication (three instead of one in the classic CG). In addition, one collective communication operation (MPI `All_gather`) is required in each iteration, to update the $\hat{\mathbf{x}}_p$ vectors (cf. Algorithm 6.1).

Algorithm 7.1 The Distributive Conjugate Gradients method (Becker, 2006, p. 160). Code executed by each processing element — *p* indices omitted for clarity reasons.

$\mathbf{p}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{L}\mathbf{x}^{(0)}$
$\hat{\mathbf{x}}^{(0)} = 0$
for $r = 0, 1, \ldots$ **do**
  $\alpha^{(r)} = \left(\mathbf{r}^{(r)}, \mathbf{r}^{(r)}\right) / \left(\mathbf{L}\mathbf{p}^{(r)}, \mathbf{p}^{(r)}\right)$
  $\mathbf{x}^{(r+1)} = \mathbf{x}^{(r)} + \alpha^{(r)}\mathbf{p}^{(r)}$
  $\mathbf{r}^{(r+1)} = \mathbf{r}^{(r)} - \alpha^{(r)}\mathbf{L}\mathbf{p}^{(r)}$
  Update $\hat{\mathbf{x}}^{(r+1)}$ {MPI `All_gather` operation}
  $\mathbf{r}^{(r+1)} = \mathbf{r}^{(r+1)} - \hat{\mathbf{H}}\left(\hat{\mathbf{x}}^{(r+1)} - \hat{\mathbf{x}}^{(r)}\right)$
  $\gamma^{(r)} = \left(\mathbf{r}^{(r+1)}, \mathbf{p}^{(r)}\right) / \left(\mathbf{r}^{(r+1)}, \mathbf{r}^{(r+1)}\right)$
  $\mathbf{p}^{(r)} = \mathbf{p}^{(r)} - \gamma^{(r)}\mathbf{r}^{(r+1)}$
  $\beta^{(r)} = -\left(\mathbf{r}^{(r+1)}, \mathbf{L}\mathbf{p}^{(r)}\right) / \left(\mathbf{p}^{(r)}, \mathbf{L}\mathbf{p}^{(r)}\right)$
  $\mathbf{p}^{(r+1)} = \mathbf{r}^{(r+1)} + \beta^{(r)}\mathbf{p}^{(r)}$
end for

7.1.2 Convergence Analysis

Becker provides some analysis and experiments on the convergence rate (§4.4.1) and numerical stability (§6.2.3) of the Conjugate Gradients method, however empirical results of only one experiment on the DCG convergence were provided (Figure 7.1).

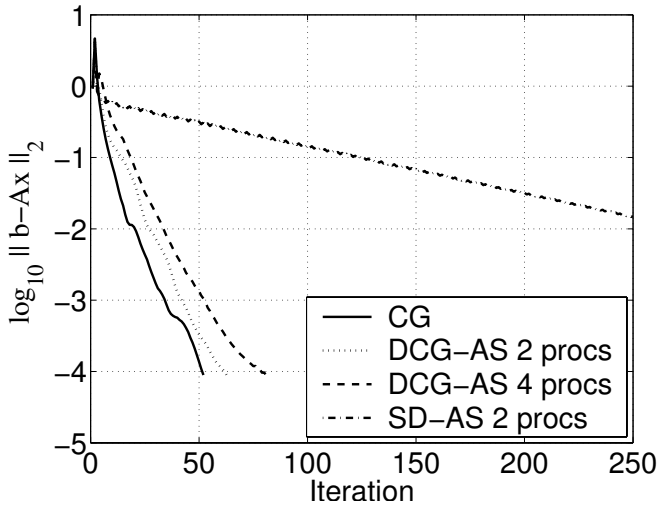


Figure 7.1: The DCG solver convergence rate for the 2D Laplace's problem with 4353 nodes and 8448 elements (Becker, 2006, p. 188). AS stands for Additive Schwarz.

In the above-mentioned experiment, the Distributive Conjugate Gradients convergence rate seems to be similar to that of the classic Conjugate Gradients method, and significantly better than that of the Distributive Steepest Descent (DSD) method[1]. However, as the number of subdomains increases the convergence rate of the DCG deteriorates. DCG running on just two subdomains requires 20% more iterations than the CG method to reach a solution with the same quality. When the number of subdomains is increased to four, 60% more iterations are required. This deterioration in the convergence rate is connected with the Additive Schwarz decomposition method rather than the DCG algorithm itself (Giraud and Tuminaro, 2006).

One of the initial project objectives was to better understand the convergence of the Distributive Conjugate Gradients method. In DCG, each processing element solves a local linear system using a CG-like method. The underlying decomposition method affects the local system by changing the right-hand side vector in each iteration. In consequence, the most important properties of the classic Conjugate Gradients method are not maintained: the residual vectors are not mutually orthogonal, and the search direction vectors are not mutually conjugate. An analysis of DCG con-

[1]The Steepest Descent method is similar to the classic CG, however the residual vector is used as a direction vector in each iteration — the direction vectors are no longer mutually conjugate.

vergence proved to be impractical and no meaningful results for upper bound were found. However, as will be shown in the following section, the convergence rate of the DBDS method is asymptotically the same as DCG and allows us to establish an analytical upper bound.

7.2 The Distributed Block Direct Solver

Analysis of the Distributive Conjugate Gradients method motivated a possible modification. Instead of performing local CG-like iterations, an exact local solution can be computed using a direct method, hence the name — *Distributed Block Direct Solver* (DBDS). The proposed modification is based on the observation that local coefficient matrices are constant and only right-hand side vectors are changing in each iteration. If a local matrix $\mathbf{L}_p$ is decomposed into triangular factors (e.g. LU or Cholesky decomposition), then an exact solution to the local system can be computed quickly using forward and backward substitutions. The pseudo-code for the DBDS method is presented in Algorithm 7.2.

Algorithm 7.2 The Distributed Block Direct Solver

1: Perform factorisation of the local matrix $\mathbf{L}$
2: **for** $r = 0, 1, \ldots$ **do**
3: $\quad \mathbf{b}^{(r)} = \mathbf{b} - \hat{\mathbf{H}}\hat{\mathbf{x}}^{(r)}$ {right-hand side vector in iteration r}
4: $\quad$ Solve $\mathbf{L}\mathbf{x}^{(r+1)} = \mathbf{b}^{(r)}$ {using forward and backward substitutions}
5: $\quad$ Update $\hat{\mathbf{x}}^{(r+1)}$ {MPI `All_gather` operation}
6: **end for**

Subsection 7.2.1 provides the time complexity analysis of the DBDS method. Then, the following two subsections investigate the convergence rate: first, the analytical upper bound for the DBDS convergence rate is derived (Subsection 7.2.2). Secondly, these results are validated with numerical experiments and compared against the DCG method convergence (Subsection 7.2.3).

7.2.1 Complexity of the Distributed Block Direct Solver

Let N denote the size of the global linear system, P denote the number of subdomains, and $n = N/P$ denote the size of local linear systems. Finally, let nnz denote

the number of non-zero elements in local matrices. It is assumed that subdomains are of the same size and contain the same number of non-zero elements. This is a reasonable assumption in the case of many matrices coming from discretisation of PDEs. If the linear system has an unbalanced structure, the size of subdomains can be adjusted (e.g. based on the number of non-zeros) to ensure good load balancing.

The cost of DBDS initialisation can be significant — the factorisation algorithms designed for dense matrices would require $\mathcal{O}(n^3)$ operations. In the case of a sparse local matrix $\mathbf{L}_p$, typically encountered when solving PDEs, the initialisation time can be significantly reduced using specialised complete factorisations, e.g. based on a multifrontal (Duff and Reid, 1983) or supernodal approaches (Demmel et al., 1999). However, the actual performance of these decompositions depends greatly on the sparsity pattern of the matrix. In consequence, it is impractical to give a definite complexity for the initial decomposition step. A more comprehensive overview and comparison of the available methods is provided in the next chapter.

Each DBDS iteration consists of a few vector operations and the solution of two triangular systems and can be completed in $\mathcal{O}(n + nnz)$ time. This is the same as the time complexity of DCG iterations. The matrix-vector multiplication by the $\hat{\mathbf{H}}_p$ matrix is performed in both methods. This operation is relatively fast since in many PDE applications $\hat{\mathbf{H}}_p$ typically has few non-zero elements.

The initialisation cost of the DCG method is $\mathcal{O}(n + nnz)$ (sparse matrix-vector multiplication), and can be significantly faster than that in the DBDS method, depending on the matrix sparsity pattern. However, the initialisation cost of the DBDS method can be amortised when a sufficient number of iterations is performed.

7.2.2 The Upper Bound for the DBDS Convergence Rate

The first step to derive an upper bound for the DBDS convergence rate, was to devise a formula describing the connection between subdomain error vectors in consecutive iterations. In addition to the nomenclature used in Subsection 7.1.1, let $\mathbf{x}_p^*$ denote the local part of an *exact* solution to the global system $\mathbf{Ax} = \mathbf{b}$, corresponding to

subdomain p, and $\hat{\mathbf{x}}_p^*$ denote the remaining elements, i.e. equation[2]

$$\mathbf{L}_p \cdot \mathbf{x}_p^* = \mathbf{b}_p - \hat{\mathbf{H}}_p \cdot \hat{\mathbf{x}}_p^* \tag{7.4}$$

is satisfied for all subproblems ($1 < p < P$). The corresponding error vectors at the r-th iteration are defined as follows:

$$\mathbf{e}_p^{(r)} = \mathbf{x}_p^{(r)} - \mathbf{x}_p^*, \tag{7.5}$$

$$\hat{\mathbf{e}}_p^{(r)} = \hat{\mathbf{x}}_p^{(r)} - \hat{\mathbf{x}}_p^*. \tag{7.6}$$

The vector $\mathbf{x}_p^{(r+1)}$ is a solution to the local system at iteration $(r+1)$, i.e. equation

$$\mathbf{L}_p \cdot \mathbf{x}_p^{(r+1)} = \mathbf{b}_p - \hat{\mathbf{H}}_p \cdot \hat{\mathbf{x}}_p^{(r)} \tag{7.7}$$

is satisfied for all subproblems ($1 \leq p \leq P$) and can be rewritten as

$$\mathbf{L}_p \cdot \left(\mathbf{x}_p^* + \mathbf{e}_p^{(r+1)}\right) = \mathbf{b}_p - \hat{\mathbf{H}}_p \cdot \left(\hat{\mathbf{x}}_p^* + \hat{\mathbf{e}}_p^{(r)}\right). \tag{7.8}$$

By subtracting Equation (7.4), Equation (7.8) can be reduced to

$$\mathbf{L}_p \cdot \mathbf{e}_p^{(r+1)} = -\hat{\mathbf{H}}_p \cdot \hat{\mathbf{e}}_p^{(r)}, \text{ and then transformed to} \tag{7.9}$$

$$\mathbf{e}_p^{(r+1)} = -\mathbf{L}_p^{-1} \cdot \hat{\mathbf{H}}_p \cdot \hat{\mathbf{e}}_p^{(r)}. \tag{7.10}$$

Applying matrix and vector norms to Equation (7.10) gives

$$\left\|\mathbf{e}_p^{(r+1)}\right\| = \left\|\mathbf{L}_p^{-1} \cdot \hat{\mathbf{H}}_p \cdot \hat{\mathbf{e}}_p^{(r)}\right\| \tag{7.11}$$

$$\leq \left\|\mathbf{L}_p^{-1} \cdot \hat{\mathbf{H}}_p\right\| \cdot \left\|\hat{\mathbf{e}}_p^{(r)}\right\| \tag{7.12}$$

$$\leq \left\|\mathbf{L}_p^{-1}\right\| \cdot \left\|\hat{\mathbf{H}}_p\right\| \cdot \left\|\hat{\mathbf{e}}_p^{(r)}\right\|. \tag{7.13}$$

Neither Equation (7.12) nor Equation (7.13) give a useful upper bound on the error reduction rate. The actual error reduction factors observed in experiments were many times smaller than values $\left\|\mathbf{L}_p^{-1} \cdot \hat{\mathbf{H}}_p\right\|$ and $\left\|\mathbf{L}_p^{-1}\right\| \cdot \left\|\hat{\mathbf{H}}_p\right\|$.

[2] If the global system has N equations and the size of the local matrix $\mathbf{L}_p$ is $n \times n$, then vectors $\mathbf{x}_p^*$ and $\mathbf{b}_p$ have n elements, vector $\hat{\mathbf{x}}_p^*$ has $(N-n)$ elements, and $\hat{\mathbf{H}}_p$ is an $n \times (N-n)$ matrix.

Equation (7.10) is however useful in establishing the upper bound based on eigenvalue analysis. Let us consider the relation between consecutive error vectors:

$$\mathbf{e}^{(r+1)} = \begin{bmatrix} \mathbf{e}_1^{(r+1)} \\ \mathbf{e}_2^{(r+1)} \\ \vdots \\ \mathbf{e}_P^{(r+1)} \end{bmatrix} \overset{(7.10)}{=} \begin{bmatrix} -\mathbf{L}_1^{-1} \cdot \hat{\mathbf{H}}_1 \cdot \hat{\mathbf{e}}_1^{(r)} \\ -\mathbf{L}_2^{-1} \cdot \hat{\mathbf{H}}_2 \cdot \hat{\mathbf{e}}_2^{(r)} \\ \vdots \\ -\mathbf{L}_P^{-1} \cdot \hat{\mathbf{H}}_P \cdot \hat{\mathbf{e}}_P^{(r)} \end{bmatrix} \quad \begin{aligned} \hat{\mathbf{H}}_i &= \begin{bmatrix} \mathbf{H}_{i,1} & \cdots & \mathbf{H}_{i,P} \end{bmatrix}, \mathbf{H}_{p,p} = 0 \\ \hat{\mathbf{e}}_i &= \begin{bmatrix} \mathbf{e}_1 & \cdots & \mathbf{e}_{i-1}, 0, \mathbf{e}_{i+1} & \cdots & \mathbf{e}_P \end{bmatrix}^T \end{aligned}$$

$$= - \begin{bmatrix} \sum_{j=1}^{P} \mathbf{L}_1^{-1} \cdot \mathbf{H}_{1,j} \cdot \mathbf{e}_j^{(r)} \\ \sum_{j=1}^{P} \mathbf{L}_2^{-1} \cdot \mathbf{H}_{2,j} \cdot \mathbf{e}_j^{(r)} \\ \vdots \\ \sum_{j=1}^{P} \mathbf{L}_P^{-1} \cdot \mathbf{H}_{P,j} \cdot \mathbf{e}_j^{(r)} \end{bmatrix}$$

$$= - \begin{bmatrix} 0 & \mathbf{L}_1^{-1} \cdot \mathbf{H}_{1,2} & \cdots & \mathbf{L}_1^{-1} \cdot \mathbf{H}_{1,P} \\ \mathbf{L}_2^{-1} \cdot \mathbf{H}_{2,1} & 0 & \cdots & \mathbf{L}_2^{-1} \cdot \mathbf{H}_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{L}_P^{-1} \cdot \mathbf{H}_{P,1} & \mathbf{L}_P^{-1} \cdot \mathbf{H}_{P,2} & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{e}_1^{(r)} \\ \mathbf{e}_2^{(r)} \\ \vdots \\ \mathbf{e}_P^{(r)} \end{bmatrix} = \mathbf{M} \cdot \mathbf{e}^{(r)} \tag{7.14}$$

To prove the convergence of the DBDS method, the following lemma will be used.

Lemma 7.1. *For every matrix* $\mathbf{M}$ *and* $\epsilon > 0$ *there exists a norm* $\|\cdot\|_*$ *such that*

$$\|\mathbf{M}\|_* \leq \rho(\mathbf{M}) + \epsilon\,. \tag{7.15}$$

Norm $\|\cdot\|_*$ *depends on both* $\mathbf{M}$ *and* ϵ*.*

Proof. Refer to Demmel (1997, pp. 279–280). ■

Theorem 7.2 (DBDS convergence). *Let* $\mathbf{M}$ *denote the matrix defined in Equation* (7.14) *and* λ_i *denote eigenvalues of* $\mathbf{M}$*. If* $\rho(\mathbf{M})$ *denotes the spectral radius of matrix* $\mathbf{M}$*, i.e.*

$$\rho(\mathbf{M}) = \max_{1 \leq i \leq n} |\lambda_i|\,, \tag{7.16}$$

then the DBDS method converges for any initial vector if and only if $\rho(\mathbf{M}) < 1$*. Furthermore,* $\rho(\mathbf{M})$ *is the upper bound for the DBDS method convergence rate.*

Proof. First, assume $\rho(\mathbf{M}) \geq 1$. Then, matrix $\mathbf{M}$ has eigenvalue $\lambda_* \geq 1$ and corresponding eigenvector $\mathbf{v}_*$. If the initial vector is selected such that $\mathbf{e}^{(0)} = \mathbf{v}_*$, then after r DBDS iterations the norm of the error vector is equal to

$$\left\|\mathbf{e}^{(r)}\right\| = \left\|\mathbf{M}^r\mathbf{e}^{(0)}\right\| = \left\|\mathbf{M}^r\mathbf{v}_*\right\| = \lambda_*^r \left\|\mathbf{v}_*\right\| = \lambda_*^r \left\|\mathbf{e}^{(0)}\right\| . \tag{7.17}$$

Since $\lambda_* \geq 1$, also $\lambda_*^r \geq 1$ regardless of the number of iterations r. In consequence, the error vector is never reduced and the DBDS method fails to converge.

Now, assume $\rho(\mathbf{M}) < 1$. To show that the convergence rate of the DBDS method is at most $\rho(\mathbf{M})$ it is sufficient to show that there exists a norm $\|\cdot\|_*$ such that

$$\left\|\mathbf{e}^{(r+1)}\right\|_* \leq \rho(\mathbf{M}) \left\|\mathbf{e}^{(r)}\right\|_* . \tag{7.18}$$

Let $\epsilon_k = (1 - \rho(\mathbf{M})) / k$. By Lemma 7.1, for every $k \in \mathbb{N}_+$ there exists a norm $\|\cdot\|_*$ such that $\|\mathbf{M}\|_* \leq \rho(\mathbf{M}) + \epsilon_k$.

Note, that

$$\rho(\mathbf{M}) + \epsilon_k = \left(\frac{1}{k} + \frac{k-1}{k}\rho(\mathbf{M})\right) \xrightarrow{k\to\infty} \rho(\mathbf{M}) . \tag{7.19}$$

By the properties of vector and matrix norms the following inequality is true:

$$\left\|\mathbf{e}^{(r+1)}\right\|_* = \left\|\mathbf{M}\mathbf{e}^{(r)}\right\|_* \leq \|\mathbf{M}\|_* \cdot \left\|\mathbf{e}^{(r)}\right\|_* \leq (\rho(\mathbf{M}) + \epsilon_k) \cdot \left\|\mathbf{e}^{(r)}\right\|_* \tag{7.20}$$

for some norm $\|\cdot\|_*$ (depending on $\mathbf{M}$ and k).

In consequence of Equations (7.19) and (7.20), there exists a norm $\|\cdot\|_*$ such that

$$\left\|\mathbf{e}^{(r+1)}\right\|_* \leq \rho(\mathbf{M}) \left\|\mathbf{e}^{(r)}\right\|_* . \tag{7.21}$$

■

Corollary 7.3. *If matrix* $\mathbf{M}$ *is symmetric, then the 2-norm satisfies Equation* (7.21) *in the proof of Theorem 7.2, i.e.*

$$\left\|\mathbf{e}^{(r+1)}\right\|_2 = \left\|\mathbf{M}\mathbf{e}^{(r)}\right\|_2 \leq \|\mathbf{M}\|_2 \cdot \left\|\mathbf{e}^{(r)}\right\|_2 = \rho(\mathbf{M}) \left\|\mathbf{e}^{(r)}\right\|_2 . \tag{7.22}$$

Let us demonstrate the applicability of Theorem 7.2 on a special case of a linear system from the discretisation of the 2D Poisson's Equation on a uniform grid using a five-point stencil. For a $n \times n$ grid and $P = n$ subdomains, the resulting matrix has $N = n^2$ rows and can be expressed in a block form (each block has $n \times n$ coefficients):

$$\mathbf{A} = \begin{bmatrix} \mathbf{L} & -\mathbf{I} & & & \\ -\mathbf{I} & \mathbf{L} & -\mathbf{I} & & \\ & \ddots & \ddots & \ddots & \\ & & -\mathbf{I} & \mathbf{L} & -\mathbf{I} \\ & & & -\mathbf{I} & \mathbf{L} \end{bmatrix}, \text{ where } \mathbf{L} = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \tag{7.23}$$

According to Equation (7.14), an $N \times N$ matrix $\mathbf{M}_{P2D}$ takes the form

$$\mathbf{M}_{P2D} = \begin{bmatrix} 0 & \mathbf{L}^{-1} & & & \\ \mathbf{L}^{-1} & 0 & \mathbf{L}^{-1} & & \\ & \ddots & \ddots & \ddots & \\ & & \mathbf{L}^{-1} & 0 & \mathbf{L}^{-1} \\ & & & \mathbf{L}^{-1} & 0 \end{bmatrix} \tag{7.24}$$

The following lemma can be used to identify the eigenvalues of matrix $\mathbf{M}_{P2D}$.

Lemma 7.4. *Let $\mathbf{A} = diag(b, a, b)$ be an $n \times n$ tridiagonal matrix ($a, b \in \mathbb{R}$). Let*

$$\theta_k = k\theta = k\frac{\pi}{n+1}\,. \tag{7.25}$$

Then, the eigenvalues of $\mathbf{A}$ can be expressed as

$$\lambda_k = a + 2b\cos\theta_k, \quad k = 1, \ldots, n, \tag{7.26}$$

and their corresponding eigenvectors are

$$\mathbf{v}_k = [\sin(\theta_k),\ \sin(2\theta_k),\ \ldots,\ \sin(n\theta_k)]^T\,. \tag{7.27}$$

Proof. For every $k = 1, \ldots, n$, the i-th inner $(1 < i < n)$ element of vector $\mathbf{A}\mathbf{v}_k$ is

$$b\sin[(i-1)\theta_k] + a\sin(i\theta_k) + b\sin[(i+1)\theta_k]. \tag{7.28}$$

Also, the first and the last elements can be expressed in this form, since the missing elements of the sum are zeros: $b\sin(0) = 0$, and $b\sin[(n+1)\theta_k] = b\sin(k\pi) = 0$.

Using the following trigonometric identity

$$\begin{aligned}\sin(\alpha-\beta)+\sin(\alpha+\beta) &= \sin\alpha\cos\beta-\cos\alpha\sin\beta+\sin\alpha\cos\beta+\cos\alpha\sin\beta\\ &= 2\sin\alpha\cos\beta\,,\end{aligned}$$

and taking $\alpha = i\theta_k$ and $\beta = \theta_k$, Expression (7.28) becomes

$$a\sin(i\theta_k)+2b\cos\theta_k\sin(i\theta_k) = (a+2b\cos\theta_k)\sin(i\theta_k)\,. \tag{7.29}$$

In consequence, for every $k = 1,\ldots,n$, the following equation is true

$$\mathbf{A}\mathbf{v}_k = \lambda_k\mathbf{v}_k, \tag{7.30}$$

i.e. λ_k are n eigenvalues of matrix $\mathbf{A}$, and $\mathbf{v}_k$ are the corresponding eigenvectors. ■

Lemma 7.5. *The eigenvalues of matrix* $\mathbf{M}_{P2D}$ *defined in Equation* (7.24) *are*

$$\lambda_{i,j} = \frac{\cos\frac{i\pi}{n+1}}{2-\cos\frac{j\pi}{n+1}}\,,\quad 1\le i,j\le n\,. \tag{7.31}$$

Proof. If $\mathbf{T} = diag(1,0,1)$ is an $n\times n$ tridiagonal matrix, then the $n^2\times n^2$ matrix $\mathbf{M}_{P2D}$ can be expressed in terms of the Kronecker product

$$\mathbf{M}_{P2D} = \mathbf{T}\otimes\mathbf{L}^{-1}. \tag{7.32}$$

Let μ_i denote the eigenvalues of matrix $\mathbf{T}$ and ν_j the eigenvalues of matrix $\mathbf{L}^{-1}$. Using Lemma 7.4 and the fact that matrix $\mathbf{T}$ is tridiagonal, $\mu_i = 2\cos(i\theta)$. Matrix $\mathbf{L}$ is also tridiagonal, hence its eigenvalues are in the form $4-2\cos(j\theta)$. The eigenvalues of the inverted matrix $\mathbf{L}^{-1}$ are its reciprocals:

$$\nu_j = \frac{1}{4-2\cos(j\theta)}\,. \tag{7.33}$$

By the mixed-product property of the Kronecker product, the eigenvalues of the matrix $\mathbf{T}\otimes\mathbf{L}^{-1}$ are the products of the eigenvalues of matrices $\mathbf{T}$ and $\mathbf{L}^{-1}$, i.e.

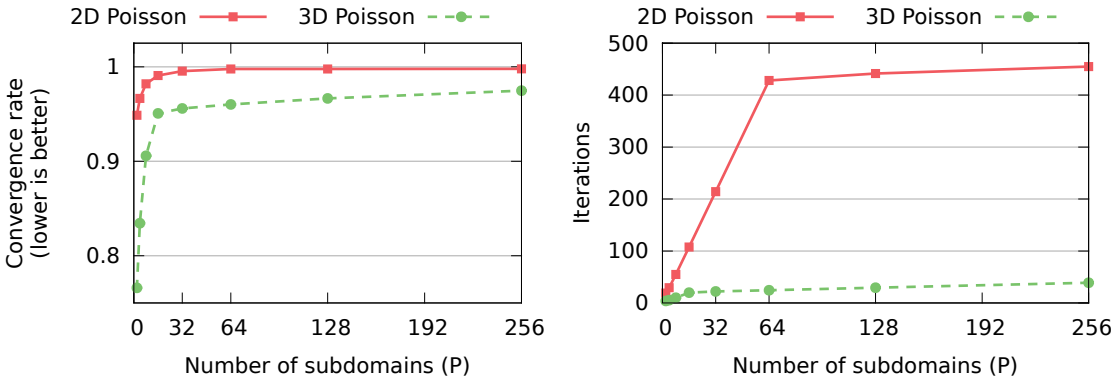
$$\lambda_{i,j} = \mu_i\nu_j = \frac{\cos\frac{i\pi}{n+1}}{2-\cos\frac{j\pi}{n+1}}\,,\quad 1\le i,j\le n\,. \tag{7.34}$$

■

In consequence of Theorem 7.2 and Lemma 7.5, the convergence rate of the DBDS method for the $N \times N$ 2D Poisson matrix and $P = n$ subdomains is bound by

$$\rho(\mathbf{M}_{P2D}) = \max_{1 \leq i,j \leq n} (|\lambda_{i,j}|) = \frac{\cos \frac{\pi}{n+1}}{2 - \cos \frac{\pi}{n+1}} = \frac{2}{2 - \cos \frac{\pi}{n+1}} - 1 \,. \tag{7.35}$$

Figure 7.2 presents how the upper bound for the DBDS method convergence changes with an increasing number of subdomains. The convergence rate deteriorates rapidly to a certain point, then further degradation is relatively slow (Figure 7.2 A). The threshold is at $P = 64$ (2D Poisson), and $P = 16$ (3D Poisson). The difference is explained by a different structure of these linear systems: the 2D Poisson matrix consists of 64 diagonal blocks of 64×64 elements, whereas the 3D Poisson matrix consists of 256 diagonal blocks of 16×16 elements.



(A) The theoretical upper bound for DBDS convergence rate.

(B) Estimated number of iterations required to reduce error norm by a factor of e^{-1}.

FIGURE 7.2: The DBDS convergence rate for Poisson matrices with 4,096 rows. When P is relatively small, the convergence rate is similar to that of the CG method, i.e. the number of iterations required to converge is proportional to $\sqrt{\kappa}$. As the number of subdomains is increased, the convergence rate deteriorates and the number of iterations becomes proportional to κ (similar to the Jacobi method). The upper bound stabilises at a certain point depending on the linear system structure.

Regardless of the number of subdomains, the convergence rate for the 3D problem is better. In consequence, the number of iterations required to reduce the norm of the error vector by an arbitrary factor is significantly lower (Figure 7.2 B). This is

connected with the condition number of the global matrix $\mathbf{A}$

$$\kappa(\mathbf{A}) = \left| \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})} \right| , \tag{7.36}$$

which is roughly 16 times greater for the 2D problem. The DBDS run with two subdomains converges in roughly four times fewer iterations for the 3D problem, which suggests a convergence rate proportional to $\sqrt{\kappa}$ — the same as in the classic CG method. As the number of subdomains increases, the difference in the number of required iterations also increases, to reach a factor of roughly 16 when $P = 64$. At this point, the DBDS method convergence is similar to the convergence of the Jacobi solver, which is an order of magnitude slower than in the CG method.

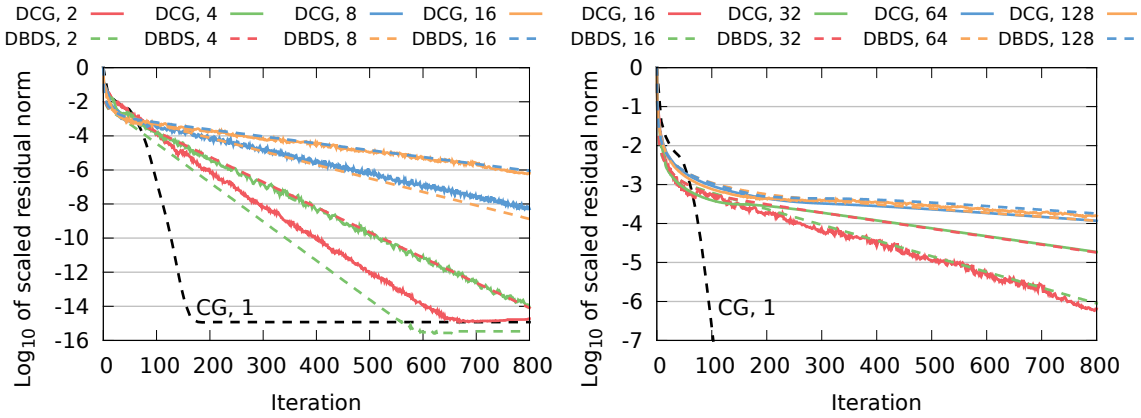
The deterioration in convergence rate is connected with the Additive Schwarz decomposition method (Giraud and Tuminaro, 2006). This effect can be reduced by using more complex *transmission conditions* allowing us to exchange more information between the subdomains (Gander, 2006; Gander et al., 2007; Loisel and Szyld, 2009). Published results indicate that the number of iterations required for convergence can be significantly reduced, however, potentially at a significantly increased cost of communication. It is not obvious how to port these methods to the GPU and how the improved transmission conditions would affect the performance in hybrid parallel platforms. Further research is required in this area.

7.2.3 Empirical DBDS Convergence

A number of experiments were performed to compare the convergence rates of DCG and DBDS methods. The results on two linear systems with 4,096 equations coming from the FDM discretisation of 2D and 3D Poisson's Equations are shown in Figure 7.3. For more details on these matrices refer to Table C.2 in Appendix C.

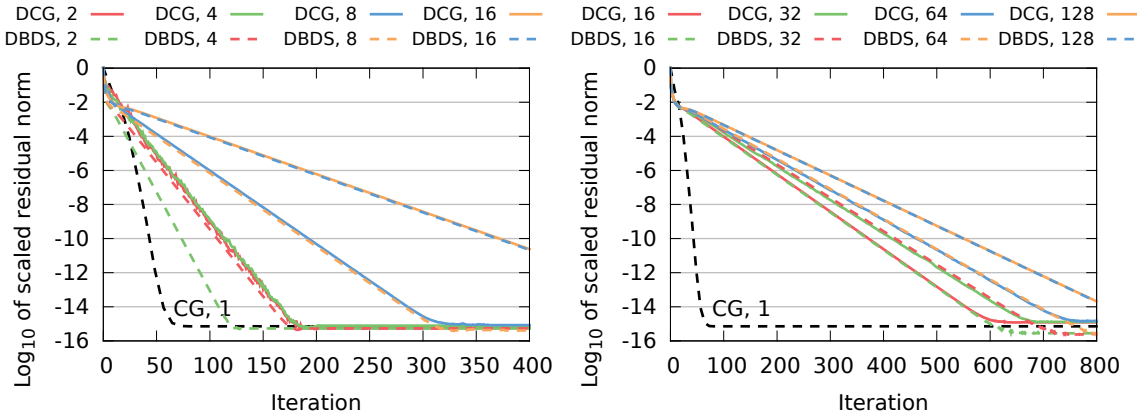
Both methods experience convergence rate degradation with increasing numbers of subdomains, caused by the underlying Additive Schwarz decomposition. For both matrices, when the number of subdomains is small the convergence of the DBDS method is better (Figures 7.3A and 7.3C). The difference becomes negligible when the number of subdomains becomes relatively large (Figures 7.3B and 7.3D).

Furthermore, the convergence trajectory of DCG tends to be jagged and irregular — there are significant differences between solutions in consecutive iterations. This



(A) 2D Poisson matrix (64^2 grid)

(B) 2D Poisson matrix (64^2 grid)



(C) 3D Poisson matrix (16^3 grid)

(D) 3D Poisson matrix (16^3 grid)

Figure 7.3: Empirical convergence of the DCG and DBDS methods with different number of subdomains. Both Poisson matrices consisted of 4,096 rows. The number of subdomains is given after the solver name. Both methods exhibit asymptotically the same convergence rate, which deteriorates with increasing number of subdomains due to the underlying Additive Schwarz decomposition. There are significant differences between solution quality in consecutive DCG iterations. In consequence, the DCG is less robust and in some cases fails to converge, e.g. 2D Poisson with 96 subdomains.

may be the source of numerical instability. Indeed, when run for the 2D Poisson matrix with 96 subdomains, the DCG method fails to converge. In contrast, the DBDS delivers smooth convergence and is more robust. Furthermore, the DBDS solver typically converges to a more accurate solution (Figures 7.3 A and 7.3 D).

The empirical results confirmed the good quality of the upper bound for DBDS derived in Theorem 7.2. The theoretical upper bound for the 4,096 × 4,096 2D Poisson matrix divided into 64 subdomains calculated from Equation (7.35) is equal to $\rho = 0.997667$. Once the convergence stabilises, the observed error norm reduction rate was equal to 0.997607, which is less than 10^{-4} lower than the upper bound.

7.3 Extensions to the DBDS Solver

The DBDS method provides a framework for the solution of linear systems, which can be adjusted to a particular problem by exchanging solver components. Some extensions that will reduce computation cost can be only used for systems with specific properties; more complex modifications can be applied to any linear system. The latter include subdomain overlapping described in Subsection 7.3.1 and the use of the mixed-precision iterative improvement scheme presented in Subsection 7.3.2.

In the general case, each local matrix is factorised with the LU method. Linear systems coming from discretisation of PDEs are often symmetric and positive-definite. In this case, the Cholesky decomposition can be used instead, providing better numerical stability, a two times faster factorisation phase, and allows us to reduce the memory footprint by a factor of two (only one triangular factor needs to be stored).

For linear systems with special structure, the factorisation may be avoided. When the number of subdomains is sufficiently large, local matrices of the decomposed Poisson matrices (2D and 3D) become tridiagonal. In this case, fast tridiagonal solvers can be used to find the solution to the local systems.

7.3.1 Overlapping

In her thesis, Becker suggested the possibility of introducing overlapping to the classic Additive Schwarz method (Figure 7.4). This is connected with additional

computation cost, however it is expected to improve the convergence rate. Bjørstad (1989) provides a case study on two subdomains, and shows that increasing the overlap resulted in a reduction in the number of iterations required for convergence.

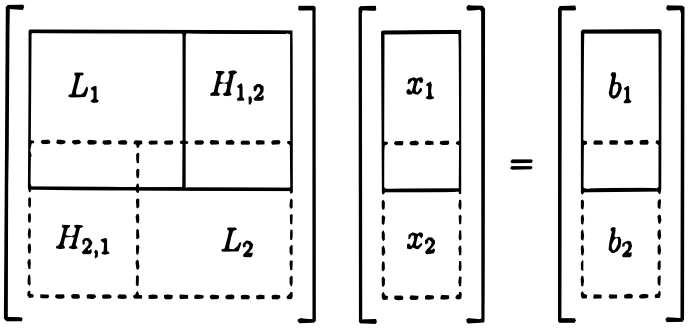


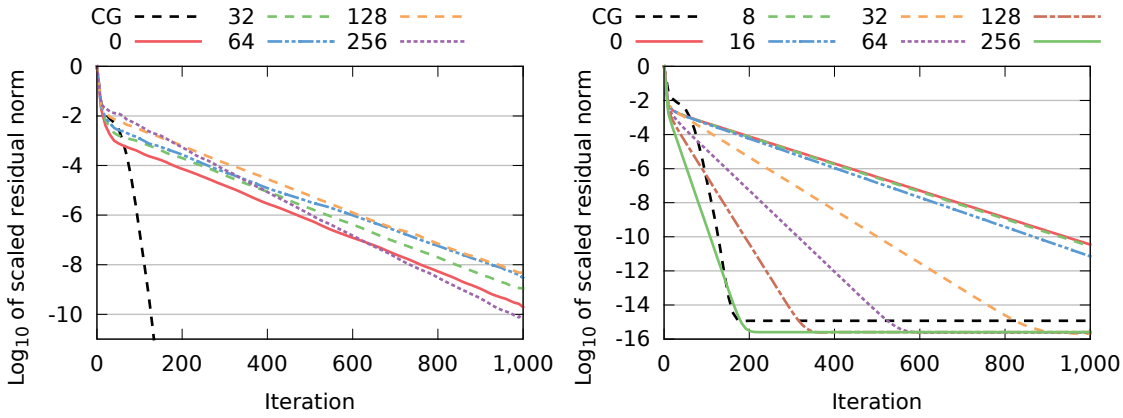
Figure 7.4: Overlapping in Additive Schwarz method (from Becker, 2006, p. 121).

In contrast, the results obtained by Becker (2006, p. 195) show that in most cases, increasing the overlap in the DCG method resulted in performance degradation. Experiments similar to those in the previous section were performed to investigate the convergence rate of overlapped DCG and DBDS methods on the 2D and 3D Poisson matrices (Figure 7.5).

The experiments confirmed that regardless of the linear system, the DCG method does not benefit from subdomain overlapping (Figures 7.5 A and 7.5 C). The results are consistent with those presented by Becker.

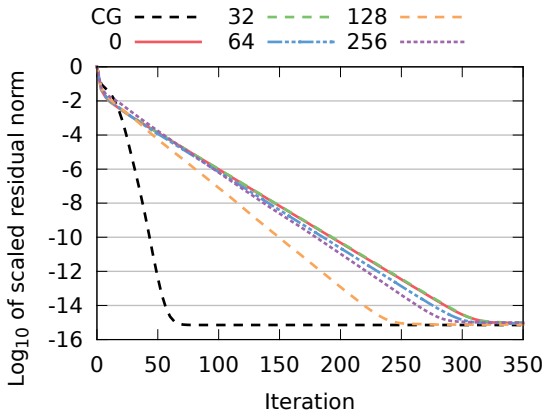
In contrast, the convergence rate of the DBDS method improves significantly with the increasing level of overlapping (Figures 7.5 B and 7.5 D). This can be explained by the way the reduction matrix $\mathbf{M}$ is changing — when the overlap is increased, the diagonal zero blocks in $\mathbf{M}$ are getting bigger. In consequence, the spectral radius of the reduction matrix is decreasing and the DBDS method converges faster.

In the case of the 2D Poisson's Equation, even using a relatively small overlap results in a reduction in the number of iterations (Figure 7.5 B). In fact, the DBDS method with an overlap of 256 converges in roughly the same number of iterations as the classic CG method. The effect of overlapping is limited in the case of the 3D Poisson's Equation. In this case, only a relatively large overlap resulted in observable improvement of the convergence rate (Figure 7.5 D). This is connected with the structure of the matrix.

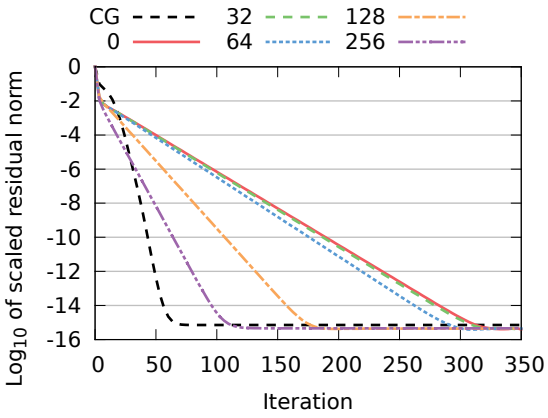


(A) 2D Poisson (64^2 grid), DCG

(B) 2D Poisson (64^2 grid), DBDS



(C) 3D Poisson (16^3 grid), DCG



(D) 3D Poisson (16^3 grid), DBDS

FIGURE 7.5: Impact of subdomain overlapping on DCG and DBDS convergence (8 subdomains). The numbers in the key denote the level of overlapping. DCG convergence plots were smoothed for clarity reasons. Unlike the DCG method (A) and (C), the convergence rate of DBDS can be significantly improved with overlapping (B) and (D), even to match the convergence of the classic CG method (B). The level of improvement depends on the structure of the linear system.

Overlapping imposes additional computation cost during the initialisation stage (larger local matrix to factorise) and at each iteration (larger right-hand side vector to solve); the communication cost is not affected. The experiments performed to determine the optimal level of overlapping in terms of time-effectiveness are presented in the next chapter.

7.3.2 Mixed-precision Iterative Improvement

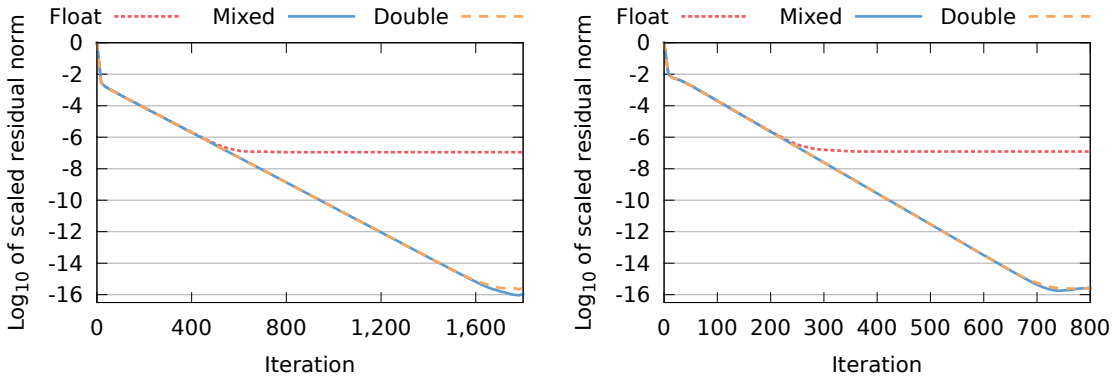
Some many-core parallel architectures (e.g. most graphics cards, IBM Cell processors) perform operations in single precision many times faster than in double precision. Mixed-precision iterative refinement (Wilkinson, 1965) aims at improving the accuracy of numerical solutions to linear systems, and enables the performance of low precision arithmetic, maintaining high precision of the final result. This technique can be beneficial for both compute- and memory-bound problems. In the former case, the instruction throughput is improved, and in the latter case, the amount of accessed data is reduced.

The key concept of employing mixed-precision iterative improvement in the DBDS method, is to perform the most demanding operations (initial factorisation, forward and backward substitutions) in single precision and only residual and solution vector updates in double precision arithmetic. The pseudo-code for the mixed-precision DBDS method is presented in Algorithm 7.3.

Algorithm 7.3 The mixed-precision DBDS method

1: Perform factorisation of the local matrix $\mathbf{L}$ {single precision}
2: **for** $r = 0, 1, \ldots$ **do**
3: $\mathbf{r}^{(r)} = \mathbf{b} - \hat{\mathbf{H}}\hat{\mathbf{x}}^{(r)} - \mathbf{L}\mathbf{x}^{(r)}$ {double precision}
4: Solve $\mathbf{L}\mathbf{z}^{(r)} = \mathbf{r}^{(r)}$ {using forward and backward substitutions; single precision}
5: $\mathbf{x}^{(r+1)} = \mathbf{x}^{(r)} + \mathbf{z}^{(r)}$ {double precision}
6: Update $\hat{\mathbf{x}}^{(r+1)}$ {MPI `All_gather` operation}
7: **end for**

The results of experiments validating the mixed-precision iterative improvement approach in the DBDS method are presented in Figure 7.6. "Float" and "Double" denote the original DBDS method in Algorithm 7.2 run with the corresponding arithmetic precision, while "Mixed" denotes the method presented in Algorithm 7.3.



(A) 2D Poisson, 8 subdomains

(B) 3D Poisson, 32 subdomains

FIGURE 7.6: The convergence of the mixed-precision DBDS method. Even though the mixed-precision method performs factorisation and triangular systems solve in single precision, it produces solutions of the same quality as the DBDS method run with double precision arithmetic. Both methods require the same number of iterations to converge.

For both problems (2D and 3D Poisson matrices), all three algorithms converge in the same manner up to a point of the single precision machine error (Figure 7.6). Even though the mixed-precision method performs factorisation and triangular systems solve in single precision, it produces solutions of the same quality as the DBDS method run with double precision arithmetic. Both methods require the same number of iterations to converge.

The mixed-precision DBDS method performs more operations than the original algorithm (sparse matrix-vector product $\mathbf{L}\mathbf{x}^{(r)}$ and two additional vector add operations). The experiments investigating whether mixed-precision iterative improvement can improve time-effectiveness of the DBDS method on various platforms are presented in the next chapter.

7.4 Conclusions

- This chapter introduces the Distributed Block Direct Solver (DBDS) — a solution algorithm for general sparse linear systems (e.g. from the discretisation of PDEs) designed to run on the hybrid CPU-GPU platforms. The DBDS solver follows the Block Relaxation scheme, but was also motivated by the

Distributive Conjugate Gradients (DCG) method (Becker, 2006). Both implementations were designed for distributed memory parallel platforms and enable straightforward hybrid MPI-CUDA parallelism. The domain decomposition can be done at the MPI level (one collective communication operation is required in each iteration), then local calculations can be executed using the numerous cores of the GPU.

- Iterations in both methods have the same time complexity. However, the DBDS solver may require significantly more expensive initialisation. When using direct sparse solvers, it is impractical to give a definite complexity for the decomposition step since it strongly depends on the sparsity pattern of the matrix. However, the DBDS initialisation cost can be amortised if a sufficient number of iterations is executed.

- The DCG method does not maintain the most important properties of the classic CG algorithm, making convergence analysis difficult. In contrast, the theoretical upper bound for the DBDS convergence rate was derived and the class of linear systems for which the DBDS method is applicable was identified.

- The numerical experiments showed trends toward the same convergence of DCG and DBDS methods, however the latter method has several advantages. The convergence of DCG is highly irregular and may become numerically unstable — in some cases the method fails to converge. The DBDS solver exhibits a smooth convergence trajectory, is more robust, and typically converges to a better quality solution than the DCG.

- Both methods experience a deterioration in convergence rate, connected with the Additive Schwarz decomposition (Giraud and Tuminaro, 2006). This effect can be reduced by using more complex transmission conditions that allow us to exchange more information between the subdomains. In consequence, the number of iterations can be significantly reduced. The possible improvements can follow the approach proposed by Gander (2006), Gander et al. (2007), and Loisel and Szyld (2009), however this is beyond the scope of this project.

- The DBDS method provides the linear systems solution framework, and can be adjusted to a particular problem. The available range of extensions include, but is not limited to, using different direct solution methods (Cholesky

factorisation, tridiagonal solver), subdomain overlapping, and mixed-precision iterative improvement scheme.

- Overlapping can be applied to both methods, however it does not improve the convergence rate of the DCG algorithm. In contrast, using overlapping subdomains in the DBDS method results in a significant reduction in the number of iterations required for convergence, even to the level comparable with the classic Conjugate Gradients method.

- The mixed-precision DBDS solver performs factorisation and the triangular systems solve in single precision, however it produces solutions of the same quality as the DBDS method run with double precision arithmetic. This can potentially lead to a significant performance improvement, especially on platforms where operations in single precision are performed many times faster than in double precision arithmetic, e.g. many GPUs.

Chapter 8

Implementation and Performance of the Distributed Block Direct Solver

This chapter investigates the numerical performance of two distributed linear solvers presented in the previous chapter: Distributive Conjugate Gradients (DCG) and the Distributed Block Direct Solver (DBDS). Both methods were implemented to run on CPU- and GPU-based parallel platforms.

The direct sparse solver is the main component of DBDS and has a strong impact on the performance, therefore experiments were conducted to select the fastest one. The overview of the available direct sparse solvers is presented in Section 8.1.

Solvers considered in this chapter were implemented using a range of third-party libraries. Section 8.2 provides the details of how the algorithms were implemented, and then integrated into the High-Performance Parallel Solvers Library (HPSL).

The results of numerical experiments on Distributive Conjugate Gradients and the Distributed Block Direct Solver on various parallel platforms are presented and discussed in Section 8.3. The conclusions are given in Section 8.4.

8.1 Direct Sparse Solvers Overview

Classic factorisations: LU and Cholesky, are impractical for large, sparse systems of linear equations due to memory requirements and the $\mathcal{O}(n^3)$ time complexity.

Another problem is that triangular factors may have significantly more non-zero elements than the original sparse matrix (fill-in). A range of specialised direct solvers addressing these issues has been proposed over the years. For an introduction to these methods refer to Duff et al. (1986).

The work performed by the direct sparse solver can be generally divided into the following three steps:

1. During the *analyse* (or *symbolic factorisation*) step, the matrix sparsity pattern is investigated to determine the data structures and a pivot sequence for efficient factorisation. In addition, matrix columns can be reordered to minimise the fill-in — this may have a significant impact on memory requirements and the speed of the solver.
2. During the *factorisation* step triangular factors are computed. This operation has to be performed only once when solving multiple linear systems with the same matrix, but different right-hand side vectors.
3. In the *solution* step the system of linear equations is solved using forward and backward substitutions and the precomputed triangular factors.

Typically, the factorisation step is the most time-consuming part of the solver (Gould et al., 2007).

The simplest implementations perform a factorisation phase similar to classic decompositions, i.e. they process consecutive rows, and immediately update the part of the matrix that has not yet been factorised. The *supernodal* solvers compute a block of rows before updating the remaining part of the matrix — this allows them to perform most of the computation using dense matrix kernels, and in consequence, improve the performance. In contrast, in the *multifrontal* approach, introduced by Duff and Reid (1983), the updates are accumulated and then propagated via already updated intermediate nodes. For more information on factorisation methods refer to the survey by Heath et al. (1991) and the textbook by Dongarra et al. (1998).

The supernodal approach for general matrices is implemented in Eigen (Guennebaud et al., 2010, `SparseLU` class) and in the SuperLU library (Demmel et al., 1999; Li, 2005). The sparse LU decomposition is also implemented in Umfpack (Davis, 2004a). Unlike the previous two methods, Umfpack follows the multifrontal strategy and

incorporates concepts presented by Davis and Duff (1997, 1999). In addition, the column pre-ordering strategy (Davis, 2004b) allows Umfpack to reduce the number of non-zero elements in the LU factors, and to give a good a priori upper bound for the fill-in, which is refined during factorisation. The Umfpack library is used as a backend for the LU decomposition and linear solver (backslash operator) in Matlab.

If the linear system is symmetric and positive definite (SPD), the Cholesky decomposition can be used instead of LU. In this case, the most notable implementation is a supernodal approach in the Cholmod library (Chen et al., 2008). In addition, Cholmod offers simple LLT and LDLT decompositions[1]. The implementations follow ideas presented by Davis and Hager (1999, 2001, 2005). In addition, Cholmod incorporates the idea of *dynamic supernodes* (Davis and Hager, 2009), which allows for further improvement in performance.

Gould et al. (2007) compared the performance of Cholmod with 10 other direct sparse solver codes. The results suggest that Cholmod typically offers very fast solution time, especially for larger problems — it was within 10% of the fastest method for 73 out of 87 matrices considered in the experiments (40 out of the 42 largest matrices). Cholmod was the fastest solver for 42 out of 87 matrices, and typically requires less memory than its competitors.

Among other direct sparse solvers for SPD problems, simple LLT and LDLT factorisations are implemented in the Eigen library. Unlike the Cholmod solvers, Eigen supports both single and double precision arithmetic. Pardiso (Schenk and Gärtner, 2004) is another library highlighted by Gould et al. (2007) for good performance. However, it is a part of the Intel Math Kernel Library and is not available for free.

8.2 Implementation Details

The High-performance Parallel Solvers Library (HPSL) comprises the algorithms considered in this thesis. The library was implemented in C++ and uses Eigen

[1]Technically, these are $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ and $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$ factorisations ($\mathbf{L}$ is a lower triangular matrix and $\mathbf{D}$ is a diagonal matrix), however in the libraries decompositions are named without superscripts, and this notation is followed in this chapter.

for basic dense and sparse linear algebra data structures and operations, and for integration with the direct sparse solver libraries listed in the previous section.

Umfpack and Cholmod are included in the SuiteSparse library (`https://www.cise.ufl.edu/research/sparse/SuiteSparse/`) and can be integrated with the Eigen library. However, they are limited to work in double precision only, and in consequence, not suitable for the mixed-precision iterative improvement extension to DBDS. The SuperLU library, which supports both single and double precision, has to be installed separately. In HPSL, the following versions of external libraries were used: Eigen 3.2.0, SuiteSparse 4.2.1, and SuperLU 4.1.

8.2.1 Hybrid Solvers

On the GPU, all linear algebra operations were performed using routines from the CUBLAS and CUSPARSE libraries. When DCG and DBDS are run on more than one GPU, local segments of the solution vector have to be exchanged in each iteration. In addition to the MPI Allgather operation, subvectors have to be downloaded from the GPU, and then the updated global solution has to be uploaded back. It is important that the CPU memory involved in CPU-GPU memory transfers is *page-locked* — this is required to overlap computation on the GPU with communication, and improves the memory throughput. Initial experiments indicated that failing to use page-locked memory results in up to 40% longer execution time.

The opportunities for computation and communication overlapping are limited. In the case of DCG, the only operation that can be performed during the exchange is the residual vector update (line 6 in Algorithm 7.1), which is typically quicker than memory transfers (Figure 8.1 A). The DBDS solver can process multiple subdomains on a single processing element (CPU core or GPU). In this case, the download of local solution vectors can be fully overlapped with computation for all but the last subdomain (Figure 8.1 B).

A number of publications on sparse direct solvers on the GPU are available (cf. Subsection 2.3.4). However, the research is focused on the factorisation step, rather than on the triangular systems solution. As will be shown later, the latter is more important for DBDS performance. Moreover, most of the published solvers are not easily available in the form of software libraries. Currently, GPU acceleration of the

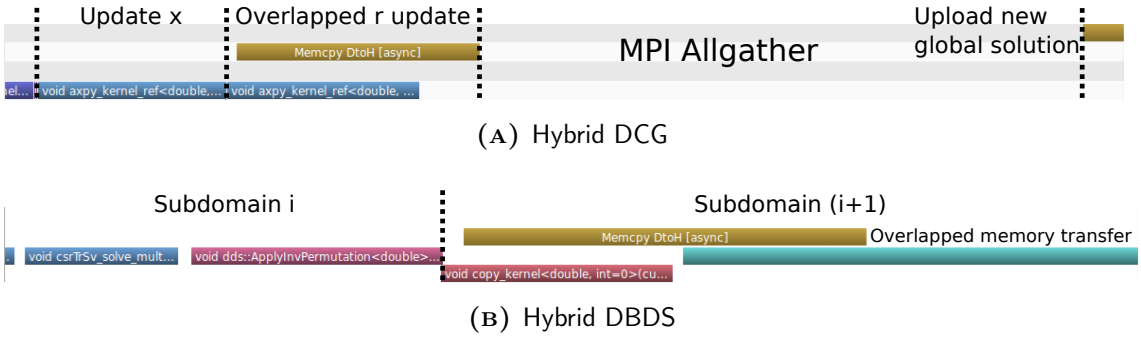


FIGURE 8.1: Computation and communication overlapping in hybrid DCG and DBDS solvers (timelines from NVIDIA Visual Profiler; yellow bars mark CPU-GPU data transfers).

factorisation step is supported in the Cholmod and BCSLIB-EXT[2] libraries, however due to limited potential performance improvement, triangular factors in the hybrid DBDS method are computed on the CPU — the Eigen LLT direct sparse solver is used for SPD matrices, and SuperLU is used in the case of general linear systems.

The solution step of a direct method involves solving two triangular systems using forward and backward substitutions. In addition, most direct sparse solver libraries produce permutations that have to be applied to a right-hand side vector before and after substitutions. These permutations allow the direct sparse solver to significantly reduce the fill-in in the triangular factors.

The solution of triangular linear systems is supported by CUSPARSE, however an analysis phase is required — data dependencies, determined by the matrix sparsity pattern, are analysed to discover parallelisation opportunities for the solution step. This analysis has to be performed once for each triangular factor.

Therefore, in addition to the actual factorisation on the CPU, the hybrid DBDS solver has to upload triangular factors and permutations to the GPU, and has to run the CUSPARSE analysis. Then, in each iteration the local linear system is solved: the right-hand side vector is permuted, two triangular solve operations are performed, and then the result is permuted again. The cost of permutations depends on the local vector size and is negligible in the total execution time.

Calling a triangular solve routine from CUSPARSE involves running multiple CUDA kernels. The number of kernels to execute strongly depends on the sparsity pattern

[2]BCSLIB-EXT is proprietary software available at `http://www.aanalytics.com/`.

of a matrix and can vary significantly, e.g. for the `parabolic_fem` matrix only three calls are required, whereas for the 2D Poisson matrix (1,048,576 equations) hundreds of kernels have to be executed for each triangular solve operation.

8.2.2 Mixed-Precision Iterative Improvement

In the mixed-precision DBDS solver, the factorisation and the solution steps are performed in single precision. The original matrix is stored in double precision, therefore it has to be cast to single precision before factorisation. Notably, the memory requirements to store triangular factors is reduced by 50%. Then, in each iteration the right-hand side vector (computed in double precision) has to be cast to single precision. After a single precision solution to the local system is found, it has to be cast to double precision before the global solution vector is updated.

On the CPU, type casting can be performed using the Eigen `cast()` template method available in classes representing vectors and matrices. On the GPU, the CUDA kernel that performs type casting had to be implemented explicitly. The cost of this operation in the total execution time is negligible.

8.3 Numerical Experiments

The experiments discussed in this section were conducted on various parallel platforms. The performance of different direct sparse solvers (Subsection 8.3.1) was compared on a machine equipped with Intel Core i7-980x (cf. Table B.1), running the Ubuntu 12.04 Server (64-bit) operating system. The main memory consisted of six 2 GB PC3-8500 DIMMs working in triple channel mode, i.e. the theoretical maximum bandwidth was 25.6 GB/s. The CPU code was compiled with `gcc` 4.6.3.

The experiments on the hybrid implementation of DCG (Subsection 8.3.2) and DBDS (Subsection 8.3.3) were performed on three machines with the same CPU, memory configuration, and software stack as in the previous paragraph. The machines were connected with a Gigabit Ethernet network and each one hosted a single Tesla C2050 GPU with the theoretical maximum memory bandwidth of 144 GB/s (cf. Table B.2). The GPU code was compiled with `nvcc` release 5.5 and CUDA

Toolkit 5.5. Moreover, the experiments on mixed-precision iterative improvement in DBDS (Subsection 8.3.4) were performed on this platform.

Finally, the experiments on subdomain overlapping (Subsection 8.3.5), and scalability and time-effectiveness of the DBDS method (Subsection 8.3.6) were performed on the Cranfield University supercomputer — Astral. It is equipped with 80 computing nodes, each consisting of two Intel Xeon E5-2660 CPUs (20 MB cache, 2.20 GHz, 8 cores) and 64 GB of main memory, i.e. each node consisted of 16 cores, and 4 GB of memory per core. The memory on each node was organised in eight 8 GB PC3-12800 DIMMs working in dual channel mode, i.e. the theoretical maximum bandwidth was 25.6 GB/s. The nodes were connected with an Infiniband low-latency network (theoretical maximum bandwidth: 40 Gbit/s).

Unless specified otherwise, all experiments were run using double precision floating-point arithmetic, and repeated ten times. The results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5%.

The matrices used in experiments come from the University of Florida Sparse Matrix Collection (Davis and Hu, 2011) and can be downloaded from `http://www.cise.ufl.edu/research/sparse/matrices/`. The 2D Poisson matrix comes from the FDM discretisation on a $1{,}024 \times 1{,}024$ grid. For more information on these matrices refer to Tables C.1 and C.2 in Appendix C. In addition, the ill-conditioned matrix was generated for the same CFD problem as described in Subsection 6.4.1, however in this section a finer mesh was used ($n = 902{,}880$, $nnz = 6{,}237{,}024$).

8.3.1 Direct Sparse Solvers

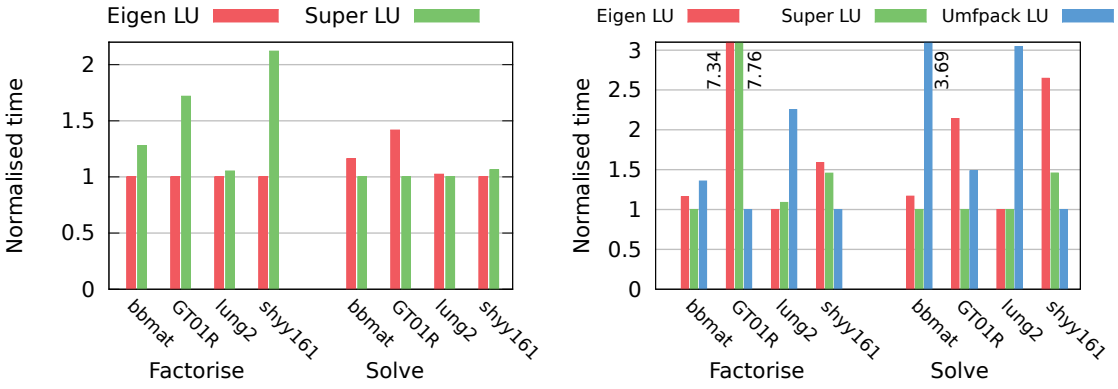
Since the direct sparse solver is an important component of the DBDS algorithm, the first step was to compare the existing methods. A number of linear systems were solved with the applicable solvers listed in Section 8.1: Eigen LU, Super LU, and Umfpack LU for general matrices. In the case of SPD matrices the following solvers were used: Eigen LLT and LDLT, and Cholmod LLT, LDLT, and Supernodal.

The experiments in this chapter are mostly focused on finding solutions to linear systems in double precision arithmetic, however the mixed-precision DBDS method

involves factorisation in single precision. In consequence, both precisions were considered in experiments on direct sparse solvers. However, Umfpack and Cholmod libraries currently do not support single precision arithmetic.

The direct solver can be divided into two steps: the symbolic and numerical factorisation performed together, and then the solution. Typically, the former require significantly more time than the latter. If the linear system has to be solved for only one right-hand side vector, it is beneficial to focus on minimising the time required for the factorisation step. However, if the same coefficient matrix appears in hundreds or thousands of linear systems differing by the right-hand side only, the cost of the solution step may become dominant. Therefore, the execution times reported in this subsection are given separately for the factorisation and solution steps.

The solution times for different matrices differ greatly, therefore to give meaningful comparison the times in Figures 8.2 and 8.3 were normalised against the fastest solver for each step. The actual execution times (normalised time equal to one) are then given in Table 8.1.



(A) Single precision (B) Double precision

FIGURE 8.2: Comparison of direct sparse solvers for general linear systems. Typically, Eigen LU is faster than Super LU during the factorisation step, but is slower during the solution step. The performance of Umfpack LU differs significantly, since it follows a different (multifrontal) factorisation method.

Only two single precision solvers are available for matrices that are not SPD. In most cases, Eigen LU offers faster factorisation, and Super LU a faster solution step, however the differences depend on the sparsity pattern of the particular system

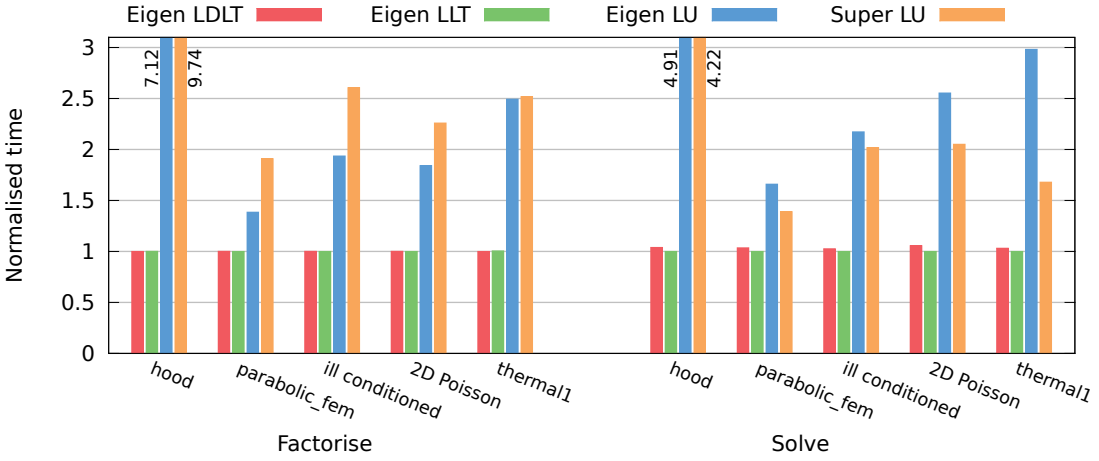
(Figure 8.2 A). The `shyy161` system is a band matrix (non-zero elements are close to the main diagonal). The `lung2` is similar, but in addition, it has several non-zero rows and columns. In consequence, the sparsity patterns of the Eigen and Super LU factors are similar, hence comparable solution step times. However, the Eigen library is more efficient when factorising a banded matrix, and is over two times faster than Super LU for the `shyy161` matrix. The other two matrices (`bbmat` and `GT01R`) have significantly more non-zero elements per row, and several non-zero elements in the corners far from the diagonal. In both cases, the factorisation is faster in Eigen, and the solution is faster in Super LU, however the differences are noticeably smaller for larger a linear system (`bbmat`).

When using double precision arithmetic, Umfpack LU becomes the third applicable direct sparse solver. As Umfpack LU follows a different (multifrontal) approach than the other two solvers, the relative differences are greater (Figure 8.2 B). The Umfpack LU factorisation step can be noticeably faster (`GT01R`, `shyy161`) or slower (`bbmat`, `lung2`) than the other two methods. Moreover, the differences between Eigen LU and Super LU factorisation times are smaller than in single precision.

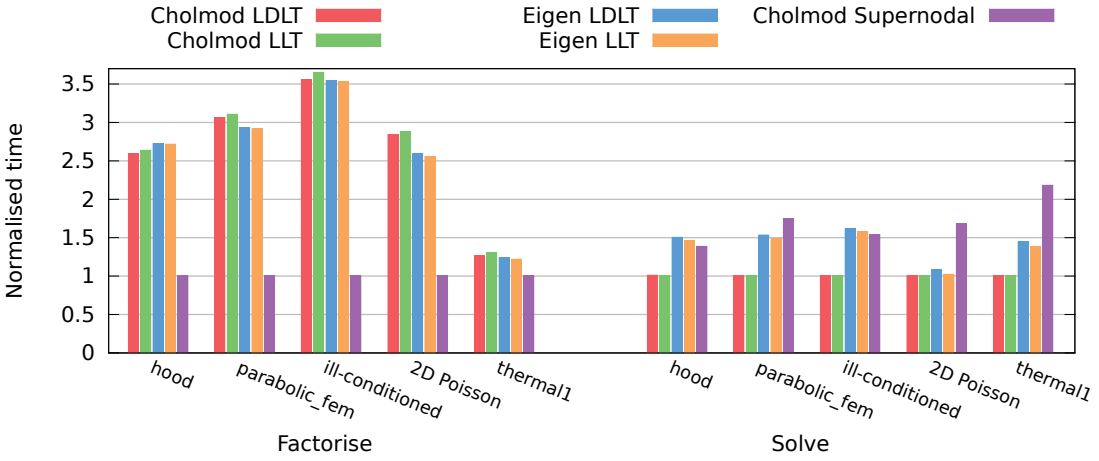
In contrast, the solution times of Eigen LU and Super LU may differ significantly in double precision (matrices `GT01R`, `shyy161`). Due to more complex factorisation in Umfpack LU, the solution step is typically slower than in the other two solvers. Moreover, Umfpack LU has significantly higher memory requirements.

In the case of SPD matrices, only Eigen LLT and LDLT solvers work in single precision. In comparison to Eigen LU and Super LU methods, solvers based on Cholesky decomposition consistently provide significantly faster factorisation and solution steps (Figure 8.3 A). The differences depend on the sparsity pattern, rather than the matrix size or number of non-zero entries — they are greatest for the `hood` matrix (relatively small, but with many non-zero elements per row), and the 2D Poisson matrix (the largest, but with the fewest non-zero elements per row). The differences in the performance between the Eigen LLT and LDLT solvers are negligible, however the former is consistently slightly faster.

The Cholmod library offers three double precision direct sparse solvers for SPD matrices. The LLT and LDLT methods are similar to their Eigen library counterparts. The differences in factorisation performance are negligible, however the Cholmod library typically offers a 30–35% faster solution step (Figure 8.3 B). This observation



(A) Single precision



(B) Double precision

FIGURE 8.3: Comparison of direct sparse solvers for SPD linear systems. Using Cholesky factorisation allows for noticeably faster factorisation and solution in comparison to more general LU decomposition (A). The differences between LLT and LDLT solvers are negligible, however Cholmod typically offers faster solution step. The factorisation in the Cholmod Supernodal method is typically much faster than in simple solvers (B), however at the cost of more complex factors, and in consequence, longer solution step.

does not hold for the 2D Poisson matrix due to its low number of non-zero elements per row. Similar to single precision, the direct LU solvers are significantly slower and they were omitted in Figure 8.3 B.

The performance of the Cholmod Supernodal solver differs significantly. The factorisation phase is typically 2.5–3.5 times faster, however it may produce more complex factors. In consequence, the solution phase is typically slower (40–120%) than in simple Cholmod factorisations. In many cases the performance of the Cholmod Supernodal solution step is comparable to that of the Eigen LLT and LDLT solvers. Since the Cholmod LDLT factorisation offers the fastest solution step, which is performed in every DBDS iteration, it was used as the direct solver in experiments discussed in the following subsections.

In most cases, the factorisation and the solution step is faster in double precision (Table 8.1). The main reason is that often the best performance was offered by specialised direct sparse solvers: Umfpack LU and Cholmod Supernodal, which are not available in single precision. However, an unexpected trend was observed when solving general matrices — in many cases the Eigen LU and Super LU solvers were faster in double precision. This phenomenon could not be observed for SPD matrices: in double precision the Eigen LLT and LDLT solvers were 20–40% slower, Eigen LU 40–70% slower, and Super LU 15–20% slower. This may suggest a design or implementation error in Eigen that limits the single precision performance for general systems, however it could not be identified.

The classic LU and Cholesky factorisations are typically impractical in the case of sparse linear systems with hundreds of thousands of equations. However, in some cases the specialised direct sparse solvers can match the performance of the iterative methods. For the `hood` matrix, the Preconditioned Conjugate Gradients method requires 140 seconds to converge to double precision machine error (Figure 6.11 C), whereas the Cholmod Supernodal solver needs only 3.1 seconds. In the case of the `parabolic_fem` matrix the corresponding solution times are: 20 seconds for the PCG (Figure 6.11 G), and 5.4 seconds for the Cholmod Supernodal solver. In contrast, for the 3D Poisson's Equation on a $64 \times 64 \times 64$ grid, the PCG requires 1.1 seconds and the fastest direct method needs roughly 140 seconds.

Table 8.1: Execution time for the best direct sparse solver for factorisation and solution steps in different precision arithmetic.

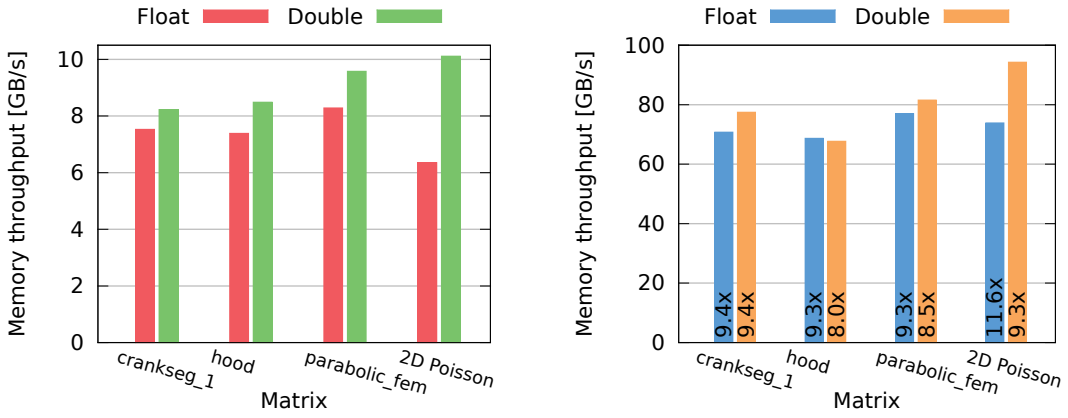
	Factorisation [s]		Solution [ms]	
Matrix	Single	Double	Single	Double
bbmat	17.88	18.25	68.3	84.0
GT01R	14.43	0.25	20.0	13.2
lung2	0.08	0.08	4.2	4.9
shyy161	5.51	0.36	37.3	9.6
hood	6.4	3.0	77.9	63.6
parabolic_fem	11.9	5.3	113.5	89.8
ill-conditioned	146.9	56.5	627.6	466.4
2D Poisson's Equation	13.2	6.7	108.2	128.5
thermal1	0.3	0.3	9.4	8.1

8.3.2 Hybrid Distributive Conjugate Gradients

The next step was to assess the benefits of running the Distributive Conjugate Gradients method on the GPU. DCG performs the same basic operations as the Conjugate Gradients method considered in Chapter 6: sparse matrix-vector multiplications, dot products, and AXPY operations. In comparison to the classic CG method, in each iteration DCG performs three SpMVs (two if run on a single subdomain) instead of one, five dot products instead of two, and five AXPY operations instead of three. Therefore, the performance model for memory bandwidth proposed in Subsection 6.4.2 needs only small adjustments to work for the DCG algorithm.

In the first experiment, DCG was run on a single subdomain for 1,000 iterations on four matrices on the Intel i7-980x CPU and the Tesla C2050 GPU. In this case, there was no need for CPU-GPU or MPI communication between iterations. The cost of uploading the input data, and downloading the result from the GPU was negligible. The observed memory throughputs are compared in Figure 8.4. Matrices

are ordered according to the number of non-zero elements per row, starting with the densest one.

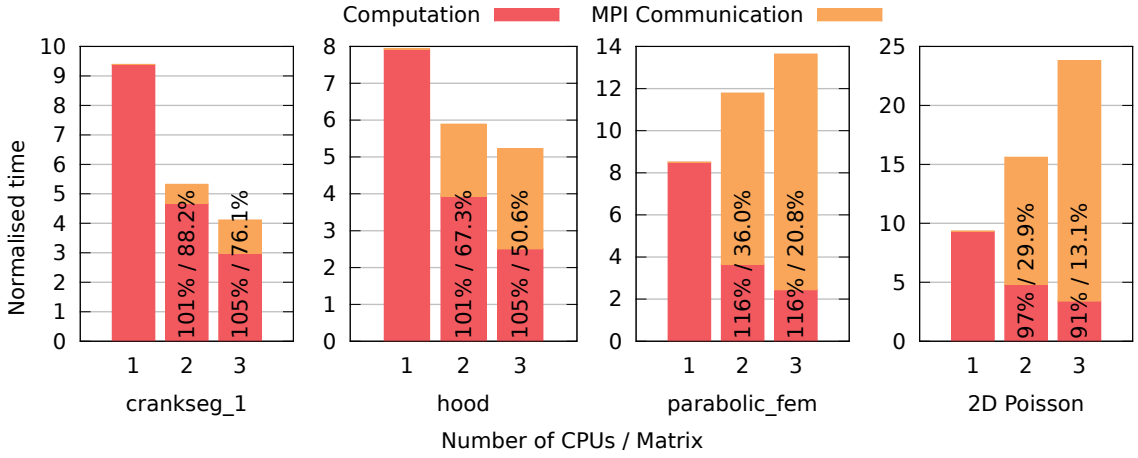


(A) Intel i7-980x (B) Tesla C2050

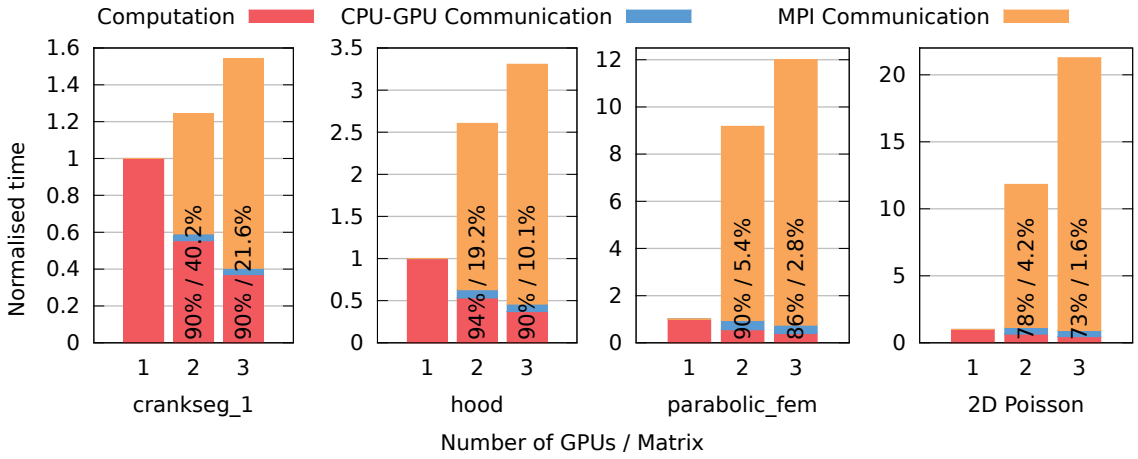
FIGURE 8.4: Performance comparison of the DCG method on the CPU and the GPU. On both platforms, the observed memory throughput does not vary significantly for different matrices and arithmetic precision. Greater differences observed for the 2D Poisson matrix on the CPU (A) are connected with a significantly higher cache hit ratio in single precision. Numbers on GPU plot bars (B) indicate the speed-up over the CPU implementation.

The memory throughput increased slightly as the average number of non-zero elements per row decreased. This is connected with the decreasing fraction of time spent on SpMV — the slowest operation in the DCG solver (cf. Subsection 6.4.4). As expected for a memory-bound problem, the differences between throughputs observed for single and double precision are small. The only exception is the 2D Poisson matrix on the CPU (Figure 8.4A). The more noticeable difference is connected with the L3 cache size. In this case, the vectors need 4 MB in single, and 8 MB in double precision — the 12 MB cache can fit only one vector in double precision leading to a higher cache miss ratio when computing dot products and AXPY operations.

The performance of DCG on the GPU is similar to that of PCG (cf. Subsection 6.4.4) and amounts to 50–65% of the theoretical peak memory bandwidth (Figure 8.4B). The numbers on plot bars denote the speed-up of the GPU implementation. The DCG solver achieved throughput of up to 94 GB/s on the GPU, and it is 9.3–11.6 times faster than the CPU implementation in single precision and 8–9.4 times in double precision.



(A) Intel i7-980x



(B) Tesla C2050

Figure 8.5: Performance of the DCG solver on multiple GPUs. Times are normalised against the time required by a single GPU for a corresponding matrix. The parallel efficiency of computing part and the whole solver are printed on plot bars. The cost of communication is more significant for sparser matrices. Gigabit Ethernet is fast enough to speed-up DCG on multiple CPUs for denser matrices, however it is too slow for the hybrid implementation — it is impractical to use more than one GPU due to communication overheads.

The next step was to run the DCG method on multiple CPUs and GPUs connected with a Gigabit Ethernet network. The experiments were performed on the same matrices and the solver was also run for 1,000 iterations. To provide meaningful comparison between different matrices, the execution times were normalised against the run on a single GPU. The results are presented in Figure 8.5. Parallel efficiencies on the plot bars were calculated separately for the computation part, and the whole

On the CPU, the speed-up of the computation part is close to linear, regardless of the linear system (Figure 8.5 A). In fact, for all but the 2D Poisson matrix super-linear speed-ups were observed. This phenomenon is connected with a higher cache hit ratio (local systems solved on each CPU are smaller), but also on the sparsity pattern of the matrix — as the number of subdomains increases, the non-zero elements move from matrix **L** (two SpMVs) to matrix **H** (one SpMV). Results confirm this analysis — the highest parallel efficiency was observed for the `parabolic_fem` matrix, where most of the non-zeros are far from the diagonal; the lowest speed-up was observed for the 2D Poisson matrix, which is a narrow band matrix.

Even though the utilisation of the Gigabit Ethernet interconnect was close to theoretical maximum (the effective throughputs observed for MPI Allgather operation were consistently around 800 Mbps), the communication overhead significantly degraded the parallel efficiency of the entire DCG solver (Figure 8.5 A). The computation-to-communication ratio decreases with matrix density. In consequence the fraction of time spent on MPI transfers is higher for problems with a low average number of non-zero elements per row. In the case of the `crankseg_1` and `hood` matrices, the communication overheads are relatively low, and DCG on multiple CPUs is noticeably faster. In the case of two sparser matrices, any benefits from faster computation on multiple CPUs are removed by the cost of MPI communication, and in consequence, using more than one CPU is impractical.

Similar to the CPU results, on the GPU the computation part of the DCG solver benefits from parallel efficiency close to 100%, however super-linear speed-ups were not observed (Figure 8.5 B) due to the relatively small global memory cache (768 KB). The parallel efficiency for the 2D Poisson matrix (with the fewest non-zeros per row) is lower since the graphics card is not fully utilised during the SpMV operation.

The hybrid DCG solver introduces an additional cost — CPU-GPU memory transfers in every iteration. The fraction of time spent on this operation is relatively

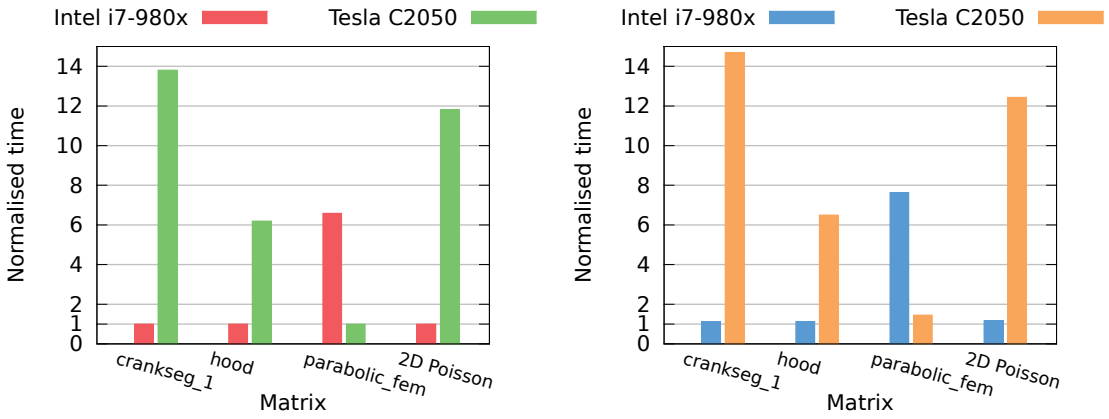
small for denser matrices, however it becomes comparable to the cost of the computation part for linear systems with 5–7 non-zero elements per row. However, this communication overhead is negligible in comparison to that connected with the MPI Allgather operation — the impact of inter-CPU data transfers on the hybrid DCG method performance is significantly more severe than in the CPU case. Indeed, it is impractical to use more than one GPU even for the `crankseg_1` matrix (200 non-zero elements per row). In the presence of Infiniband interconnect, the cost of MPI communication would become comparable to the cost of CPU-GPU transfers. In this case, it would be possible to obtain high parallel efficiency with the hybrid DCG solver run on multiple GPUs.

8.3.3 Hybrid Distributed Block Direct Solver

Experiments similar to those described in the previous subsection were conducted for the hybrid DBDS implementation. This method also performs sparse matrix-vector multiplications and AXPY operations, however the main cost is associated with the factorisation and the solution step of the underlying direct sparse solver. For the latter operations, it is not possible to derive performance models similar to those for CG and DCG. Therefore, the execution times were measured instead of memory bandwidth. Each matrix was decomposed into four subdomains.

In the hybrid DBDS implementation the factorisation step is performed on the CPU (cf. Subsection 8.2.1), therefore its performance is the same as in the CPU-only implementation. In consequence, only times of the solution step of the DBDS method are reported. Due to significant differences between the matrices, the execution times were normalised against the fastest run for each problem (the base time for single and double precision is the same). The results are presented in Figure 8.6.

Regardless of the linear system and precision used, the differences between the CPU and GPU implementations are significant (the slower method runs 6–14 times longer). In most cases, the CPU implementation is faster. However, the GPU solver is 6.5–7.5 times faster for the `parabolic_fem` matrix, depending on arithmetic precision. Indeed, this is the only linear system where the difference in execution time in double precision is noticeably (45%) longer than in single precision. For all the other matrices the difference does not exceed 15%.



(A) Single precision (B) Double precision

FIGURE 8.6: Performance comparison of the solution phase of the DBDS method on the CPU and the GPU. Times are normalised against the fastest method (the base time is the same for single and double precision). The triangular solve performance on the GPU strongly depends on parallelisation opportunities determined by data dependencies from the sparsity pattern. In consequence, the DBDS performance on the GPU is highly irregular.

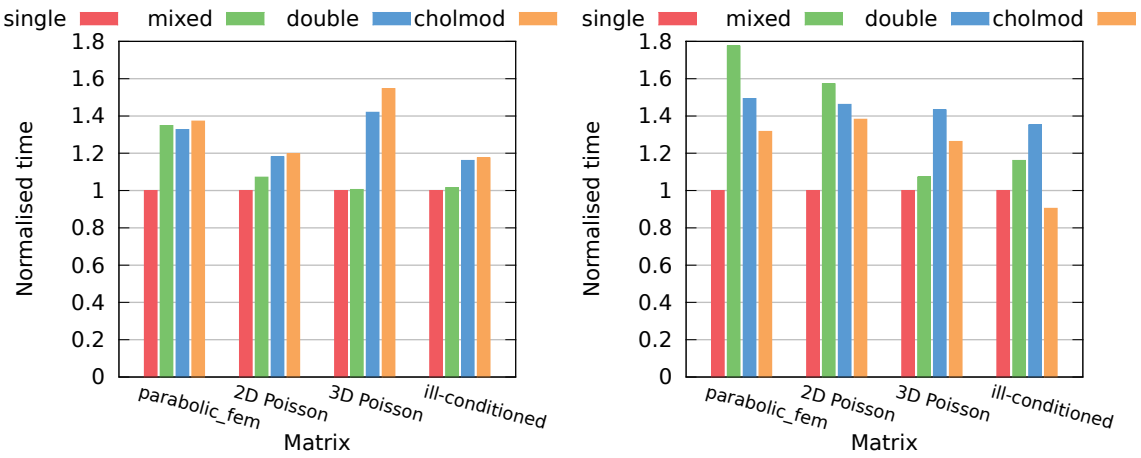
The GPU implementation was investigated further with the NVIDIA Visual Profiler. The analysis indicated a highly irregular performance when solving the sparse triangular systems. On the GPU, this operation is performed in multiple CUDA kernel calls. The number of calls strongly depends on the sparsity pattern of the matrix: in the case of the `parabolic_fem` matrix, only three kernels are required, whereas for the 2D Poisson matrix 1,019 calls were needed due to data dependencies limiting parallelisation opportunities. In the latter case, the performance is not limited by instruction or memory throughput, but by overheads associated with CUDA kernel execution. This also explains why the differences between single and double precision execution times are significantly lower for `crankseg_1`, `hood`, and 2D Poisson matrices.

Due to significant MPI communication overheads over Gigabit Ethernet network (cf. Subsection 8.3.2), and limited performance improvement from solving triangular linear systems on the GPU, running the DBDS method on multiple GPUs is impractical — no improvement could be observed in comparison to the runs on a single GPU and multiple CPUs. Therefore, the results of the hybrid DBDS method on multiple GPUs are omitted.

8.3.4 Mixed-Precision Iterative Improvement

The next experiment was focused on the applicability of mixed-precision iterative improvement in DBDS. The algorithm was run for 1,000 iterations on four SPD matrices on Intel i7-980x CPU and Tesla C2050 GPU. Since the Cholmod library does not support single precision, in this experiment Eigen LLT was used instead.

Figure 8.7 presents the execution time of the DBDS solver run in single and double precision, and with mixed-precision iterative improvement. The results on DBDS with Cholmod LDLT factorisation in double precision are given for reference. Due to significant differences for various linear systems, the execution times were normalised against the time of the single precision solver.



(A) Factorisation step (B) Solution step

FIGURE 8.7: Performance of mixed-precision DBDS on the CPU. Execution times were normalised against the single precision solver for each matrix. Benefits from using mixed-precision are greater when triangular factors are more complex. Otherwise, mixed-precision DBDS may be slower than original DBDS in double precision due to various overheads. Cholmod does not support single precision, thus slower Eigen LLT was used. Cholmod may be faster than Eigen LLT in single precision, rendering mixed-precision DBDS impractical.

The first two matrices (`parabolic_fem` and 2D Poisson) are relatively easy to factorise and the level of fill-in is small. In contrast, the factorisation of the 3D Poisson and the ill-conditioned matrices require significantly more time and the factors are more complex. This had a direct reflection in relative execution times observed for

different precisions on the CPU — the double precision factorisation step is 20–40% slower (Figure 8.7 A) and the solution step is 40–50% slower (Figure 8.7 B).

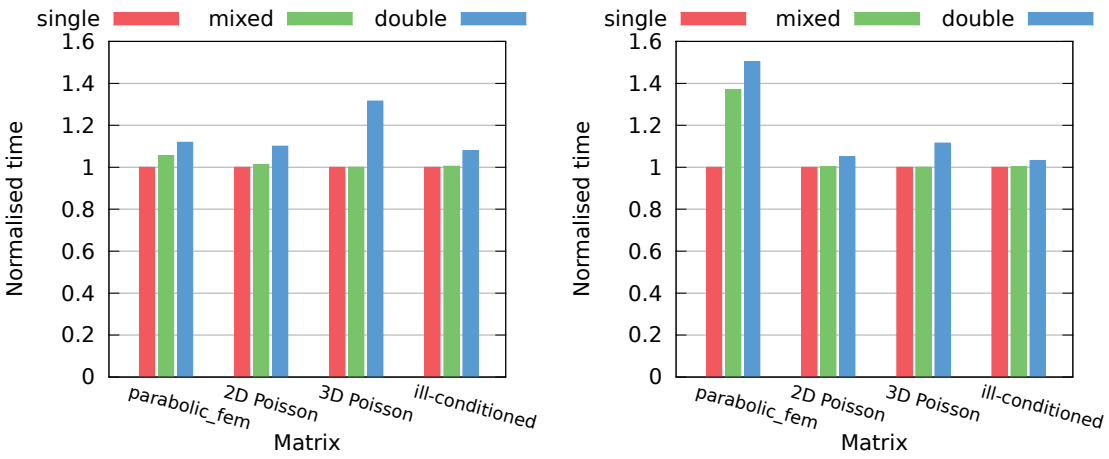
The factorisation step of the mixed-precision solver is performed in single precision, however the original matrix is stored in double precision and has to be cast to single precision. This overhead is generally negligible, however it may become significant for matrices that are easy to decompose, e.g. `parabolic_fem` (Figure 8.7 A). The time required for double precision Cholmod LDLT decomposition is comparable to that of double precision Eigen LLT.

Each iteration of mixed-precision DBDS involves additional SpMV and AXPY operations. Similar to the factorisation step, this overhead becomes significant if the triangular factors have few non-zero elements. In fact, the mixed-precision solution step is slower than in double precision for the `parabolic_fem` and 2D Poisson matrices (Figure 8.7 B). When triangular factors are more complex, the mixed-precision iterative improvement allows us to reduce the time spent on the solution phase by 10–30%. However, the use of mixed-precision DBDS may be impractical on the CPU, since the Cholmod LDLT solution step is typically faster than in the Eigen LLT. Indeed, in the case of the ill-conditioned matrix, DBDS with the Cholmod direct solver in double precision is faster than Eigen LLT in single precision.

Furthermore, the mixed-precision solver may become numerically unstable if the matrix has too high a condition number (Langou et al., 2006; cf. Subsection 2.3.4). Indeed, that is the case with the ill-conditioned matrix — the mixed-precision DBDS breaks down and returns an incorrect result.

Differences in DBDS performance are smaller on the GPU (Figure 8.8). Even though the factorisation is done in single precision, there are other overheads: the factors have to be transferred to the GPU, and they have to be analysed before the triangular solve can be performed. The former operation is 50% faster in the mixed-precision solver, however the latter does not depend on the precision but rather the sparsity pattern of the factors. In consequence, the performance of factorisation in the mixed-precision solver is close to that of DBDS in single precision, and offers 10–30% improvement over the factorisation in DBDS in double precision (Figure 8.8 A).

The differences in the solution step for three matrices are negligible (Figure 8.8 B). In this case, the triangular solve operation involves hundreds or even thousands of



(A) Factorisation step

(B) Solution step

FIGURE 8.8: Performance of mixed-precision DBDS on the GPU. Execution times were normalised against the single precision solver for each matrix. For many linear systems DBDS performance does not vary significantly for different precisions due to limited parallelisation opportunities in triangular solve — hundreds of CUDA kernels must be called in each iteration, and kernel execution overheads become the bottleneck of the GPU solver.

CUDA kernel calls and the performance is limited by kernel execution overheads rather than instruction or memory throughput, thus does not depend on precision (cf. Subsection 8.3.3). In contrast, the solution step for the `parabolic_fem` matrix is memory-bound and is 50% slower in double precision. In this case, the mixed-precision DBDS solver is faster, however due to additional operations (SpMV, AXPY) overhead the performance is improved by 10% only.

It is important to note that the mixed-precision DBDS solver does not offer any savings in CPU-GPU and MPI communication — parts of the global solution exchanged in each iteration have to be stored in double precision to ensure that the linear system is solved to double precision. In contrast, the single precision solver reduces the amount of data exchanged at each iteration by 50%.

8.3.5 Subdomain Overlapping

Subdomain overlapping allows us to significantly reduce the number of iterations required by DBDS. This subsection presents results on how the number of subdo-

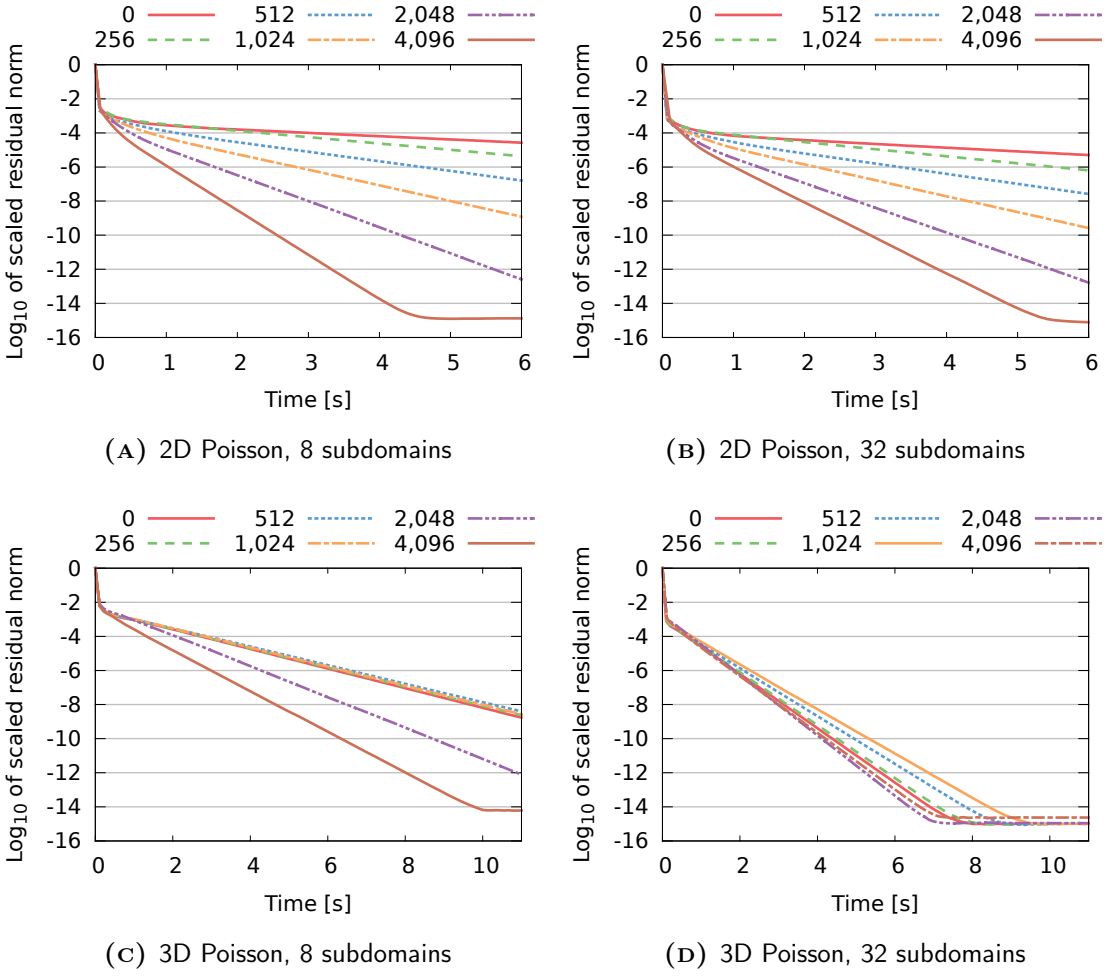
mains and the level of overlap affect time-effectiveness of the DBDS method. The experiments were performed on Poisson matrices with 262,144 equations: the FDM discretisation on a 512×512 grid in the 2D case, and on a $64 \times 64 \times 64$ grid in the 3D case (cf. Table C.2 in Appendix C). This way the results in this subsection complement the results presented in Subsection 7.3.1.

Figure 8.9 shows how time-effectiveness of the DBDS method changes with the level of overlap (in this case, the cost of initial factorisation was omitted). In the case of 2D Poisson matrix, the performance noticeably improves with the level of overlap, regardless of the number of subdomains (Figures 8.9 A and 8.9 B). However, the time required for convergence increased when the number of subdomains was increased from 8 to 32, suggesting limited scalability of the DBDS method (each subdomain can be handled independently by separate processing elements).

In contrast, time-effectiveness of the DBDS method on 3D Poisson matrix is not so tightly connected with the level of overlap. On eight subdomains, the improvement can only be observed when the overlap is at least 2,048 rows (Figure 8.9 C). No significant differences could be observed when the DBDS method was run on 32 subdomains (Figure 8.9 D). However, in this case the solver on 32 subdomains is faster than on 8 subdomains — the scalability of the DBDS method is likely to be better for the 3D matrix.

As was shown in the previous subsection, the cost of factorisation can be significant, and can be further increased by overlap, since the local problem matrix becomes bigger. Therefore, experiments taking that cost into account were conducted. Figure 8.10 presents the total time required by the DBDS method with changing number of subdomains (CPU cores) to converge to double precision machine error. Very large overlaps (up to 65,536 rows) were included in these experiments.

Further increase in the level of overlap had a positive impact on DBDS time-effectiveness for the 2D Poisson matrix (Figure 8.10 A). As expected, in this case the scalability of the method is limited — the best execution time was recorded on just four cores. However, in this case DBDS offers a significant improvement over the PCG method. The latter requires 4.7 seconds on a single core to converge, whereas the DBDS needs 1.2 seconds on four cores (almost linear speed-up). Notably, the direct solver was over two times faster than the PCG method.



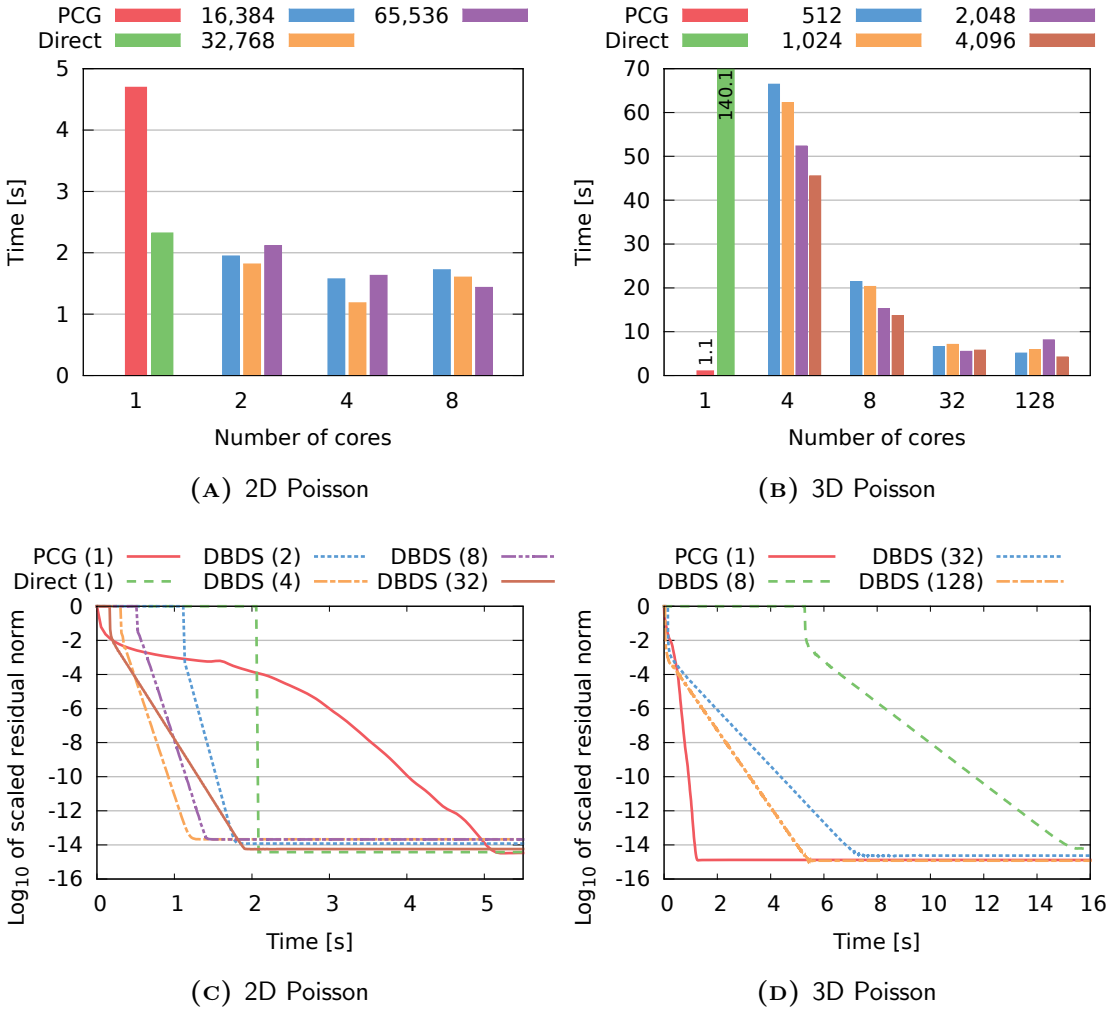
(A) 2D Poisson, 8 subdomains

(B) 2D Poisson, 32 subdomains

(C) 3D Poisson, 8 subdomains

(D) 3D Poisson, 32 subdomains

Figure 8.9: Time-effectiveness of DBDS with overlapping subdomains (factorisation time was excluded). The performance depends on properties of the matrix — the time to convergence noticeably decreases with increasing overlap for the 2D Poisson matrix, but the effect is limited for the 3D Poisson matrix.



(A) 2D Poisson

(B) 3D Poisson

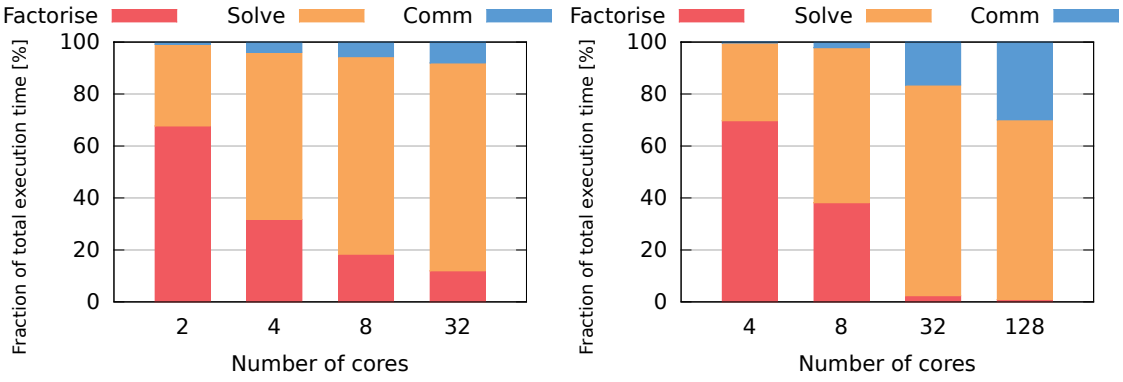
(C) 2D Poisson

(D) 3D Poisson

FIGURE 8.10: DBDS time and convergence trajectory to reach double precision accuracy (factorisation time was included). In (C) and (D), the number in parentheses denotes the number of cores. DBDS performance strongly depends on the sparsity pattern of the matrix — for the 2D Poisson matrix DBDS compares favourably with the PCG method.

In contrast, the PCG solver requires only 1.1 second to solve the 3D Poisson matrix, whereas the direct sparse solver needs an enormous amount of time (Figure 8.10 B). The difficult factorisation makes the DBDS method slow when the number of subdomains is small (2–8). In this case, the performance improves with increasing overlap due to significant reduction in the number of iterations and relatively low additional factorisation cost. As the number of subdomains is further increased, the cost of factorisation, and thus the execution time, is significantly reduced. However, there is no significant improvement between 32 and 128 cores, because any reduction in computation time is balanced out by the increasing cost of MPI communication. In consequence, even on 128 cores the DBDS method requires 4.2 seconds to converge, almost four times the time needed by the PCG solver run on a single core.

When the number of cores is small, the execution is dominated by the factorisation step (Figure 8.11). As the number of subdomains (cores) is increased, the factorisation is performed on smaller matrices with more favourable sparsity pattern. The solution step is performed on smaller local systems, however the convergence rate deteriorates and more iterations are required. In consequence, the fraction of time spent on factorisation decreases.



(A) 2D Poisson (32,768 overlap) (B) 3D Poisson (4,096 overlap)

Figure 8.11: Execution time breakdown for the DBDS solver. Increasing the number of cores (subdomains) results in local linear systems that are easier to factorise, however the number of iterations required to reach the desired tolerance is also increased. Moreover, when the number of cores is high, the communication overhead becomes noticeable.

In the case of the 2D Poisson matrix, the factorisation step becomes relatively cheap ($< 30\%$) on just four cores (Figure 8.11 A). In contrast, for the 3D Poisson matrix,

this step requires a significant fraction ($\sim 40\%$) of the total execution time even on eight cores (Figure 8.11B). That cost becomes negligible when 32 cores are used, however in this case, the cost of communication becomes noticeable.

8.3.6 Scalability and Time-Effectiveness of DBDS

In the final experiment, the scalability and time-effectiveness of the optimised DBDS method was assessed on the large linear system: 2D Poisson's Equation on a 2,048 × 2,048 grid (n = 4,194,304, nnz = 20,963,328). In the previous subsection, the optimal level of overlap was $n/8$, therefore in this subsection it is set to 524,288 rows. This may limit the scalability when the number of subdomains is large, when the local systems are small in comparison to the overlap, however this setup allows DBDS to converge quickly. Since factors of the 2D Poisson matrix are relatively simple, there is little gain from using mixed-precision iterative improvement (cf. Subsection 8.3.4), therefore the classic double precision implementation was used.

The optimised DBDS method was compared against the DCG and the DBDS without overlapping. All methods were run for 10,000 iterations on 1, 2, 4, 8, 16, 32, 64, and 128 Astral cores. The parallel efficiency was calculated separately on execution times required to perform a fixed number of iterations, and on times required to reach the desired error tolerance. The results including and excluding the communication cost are given separately. The results are presented in Figures 8.12 (DCG), 8.13 (DBDS), and 8.14 (optimised DBDS). The solution error tolerance and the number of iterations required to reach it is given in parentheses.

The efficiency of the DCG method decreases quickly with the increasing number of cores (Figure 8.12) due to two main factors. The first is that the communication cost (MPI Allgather operation) is increasing. Indeed, the biggest drop in parallel efficiency can be observed between 8 and 16 cores — the physical CPU consists of eight cores, thus up to this point most of the exchanged data could be fetched from the L3 cache. The second, and perhaps more important factor, is the need to subtract old and new *global* solution vectors in each iteration (line 8 in Algorithm 7.1). The cost of this step is constant, regardless of the number of cores, and limits the scalability of the DCG solver.

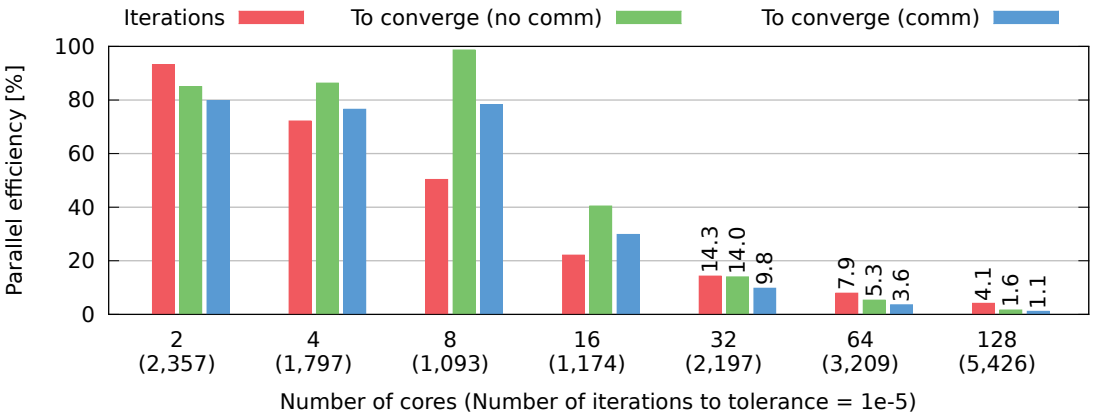
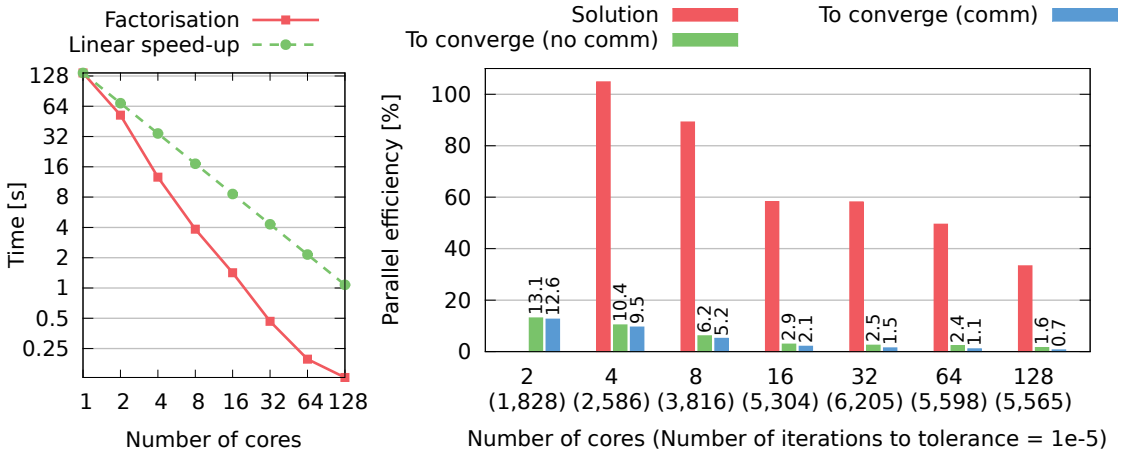


Figure 8.12: Parallel efficiency of the DCG method on Astral. Rapidly decreasing parallel efficiency of the computation part of DCG is connected with global vector subtraction, which does not scale with increasing number of subdomains. The impact of communication overhead is noticeable, but not crucial for DCG performance.

Unexpectedly, the number of iterations required by DCG to reduce the scaled residual vector norm below 10^{-5} is initially decreasing with the increase in the number of subdomains. This phenomenon demonstrates the highly irregular convergence of the DCG method, and can no longer be observed if higher accuracy is desired. Furthermore, it is not possible to predict the optimal number of subdomains and the number of iterations. In this case, the minimum is observed for eight cores, and indeed the time required to reach the desired tolerance (including communication cost) is roughly 6.5 times shorter than on one core.

Increasing the number of cores (subdomains) in the DBDS method had a completely different impact on the factorisation and solution steps performance. As the number of subdomains increased, the sparsity pattern of the local linear systems became more favourable for decomposition. In consequence, the time required for the factorisation step is decreasing much faster than a linear speed-up, e.g. on 128 subdomains, it is roughly 8 times shorter than would be expected with 100% parallel efficiency (Figure 8.13 A). Therefore, in some cases it may be beneficial to run the DBDS solver with more subdomains than the actual number of available cores, however this strongly depends on a particular linear system.

On a single subdomain, the DBDS method is reduced to running the direct sparse solver on the entire matrix. In consequence, only one iteration is performed in the



(A) Factorisation step

(B) Solution step

FIGURE 8.13: Parallel efficiency of the DBDS method on Astral. The reduction in the factorisation step time is faster than the increase in the number of cores (A) — the sparsity pattern of local matrices becomes more favourable for decomposition. Parallel efficiency of the computation part in DBDS is significantly better than in DCG. However, the number of iterations required to reach the desired tolerance grows fast with an increasing number of subdomains, rendering non-overlapped DBDS solver impractical.

solution step. Therefore, the solution time for two subdomains was used as a base to calculate the parallel efficiency. When run for a fixed number of iterations, the parallel efficiency of the DBDS is much higher than that of the DCG method (Figure 8.13B) since all steps of a DBDS iteration scale well (there is no need to calculate the difference between consecutive global solution vectors). Indeed, a super-linear speed-up was observed on four cores due to smaller local systems that allowed for a higher cache hit ratio. However, the parallel efficiency decreases with the increasing number of cores due to communication overheads. Similar to the DCG method, the most significant drop could be observed between 8 and 16 cores.

The number of iterations required by non-overlapped DBDS to reach a predefined error tolerance increases quickly with the number of subdomains. In consequence, the parallel efficiency calculated on execution times to find a solution with a specified accuracy are very low (Figure 8.13B). Indeed, in this case using the DBDS method on more than one core is impractical — it is faster to run a direct sparse solver.

The results are different when using the optimised DBDS method (Figure 8.14).

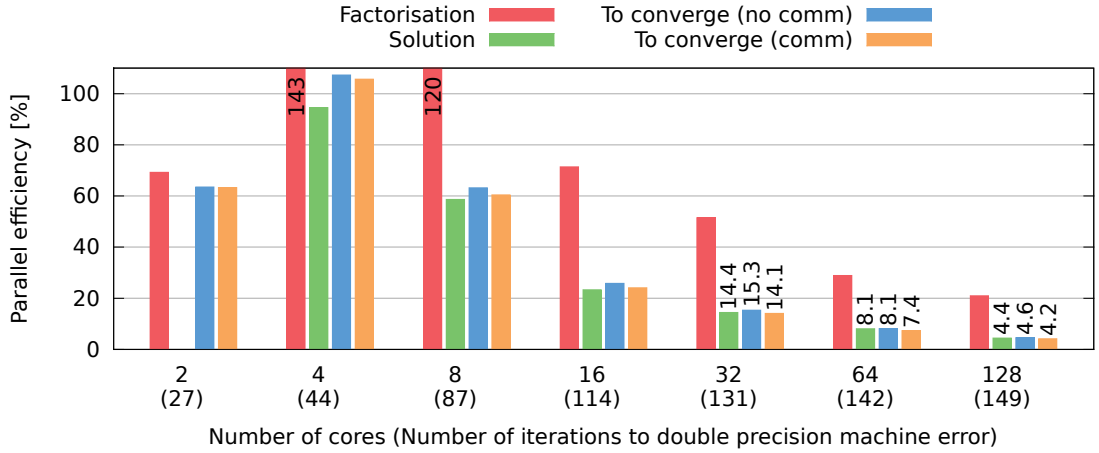


Figure 8.14: Parallel efficiency of the optimised DBDS method (overlap = 524,288) on Astral. Parallel efficiency is lower than in non-overlapped DBDS, however the number of iterations required to converge is reduced by a few orders of magnitude.

The speed-ups observed for factorisation step are significantly lower due to high level of overlap. The increase in parallel efficiency on four cores is due to the more favourable sparsity pattern of local linear systems. When the number of subdomains is increased further, the reduction in time required for the factorisation step is limited due to the relatively small decrease in the local matrix size (for $n \geq 8$ the overlap is bigger than the original local linear system). In fact, this also affects the solution step (larger triangular systems to solve in each iteration), leading to parallel efficiency degradation. In consequence, when run for a fixed number of iterations, the scalability of the optimised DBDS method is limited.

However, using a big overlap greatly improves the convergence rate — the optimised DBDS method calculates a solution accurate to the double precision machine error in less than 150 iterations (Figure 8.14). In consequence, the fraction of time spent on MPI communication is reduced. Indeed, the impact of communication overhead on execution time (thus parallel efficiency) is minimal.

The great reduction in the number of iterations makes optimised DBDS the fastest method for the 2D Poisson problem (Figure 8.15). The best time-effectiveness could be observed when the optimised DBDS method was run on 8 and 128 cores — the solver required roughly 26 seconds to compute the solution. In comparison, the PCG method requires 350 seconds and the fastest direct sparse solver requires

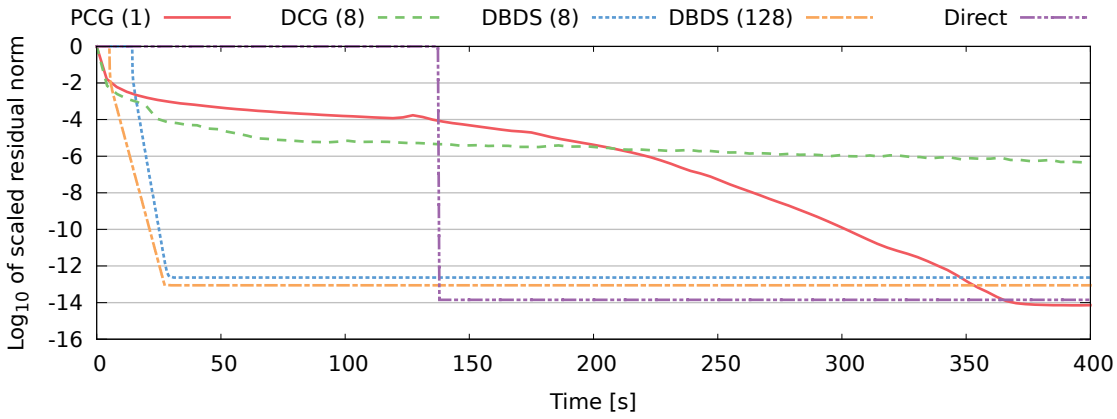


FIGURE 8.15: Comparison of the various solution methods on Astral (2D Poisson's Equation on 2,048 × 2,048 grid). The DBDS method run with overlap of 524,288 rows allows us to solve the problem 13.5 times faster than with the PCG method and 5.4 times faster than the fastest direct sparse solver.

roughly 140 seconds (both run on a single core). In the case of 2D Poisson's Equation on $2{,}048 \times 2{,}048$ grid, the optimised DBDS method on eight cores is 13.5 times faster than the PCG method (168% parallel efficiency), and 5.4 times faster than the direct sparse solver (67% parallel efficiency).

8.4 Conclusions

This chapter presents and discusses the results of numerical experiments on the DCG and the DBDS methods on two parallel platforms: a large CPU cluster with Infiniband interconnect and a multi-GPU cluster with Gigabit Ethernet network.

A direct sparse solver is the main component of the DBDS method and has a significant impact on the solver performance. Therefore, the first step was to compare a range of existing state-of-the art implementations.

- Experiments indicated that the Cholmod Supernodal implementation offers the fastest factorisation for SPD matrices. LLT and LDLT decompositions implemented in the Cholmod and Eigen libraries typically spend 2.5–3 times longer on the factorisation step, however they reduce the time required for

the solution step by 30–55%. Since the DBDS method performs the latter operation many times, the additional cost of factorisation is quickly amortised.

- Typically, the solution step of the Cholmod implementation is faster than that in the Eigen library. In consequence, in the case of DBDS, Cholmod LDLT is the best choice for a direct sparse solver.

- The time required for the factorisation step in a direct solver greatly depends on the sparsity pattern of the linear system, rather than on the matrix size or the number of non-zeros. The execution time required by a direct solver, and in consequence, time required by DBDS, cannot be easily predicted.

The next step was to investigate the performance of hybrid DCG and DBDS implementations on a multi-GPU cluster.

- In contrast to DBDS, the DCG algorithm consists of operations that allow us to accurately model the performance in terms of memory throughput. The DCG method achieves performance similar to that of the PCG solver on both CPU (6.5–10 GB/s; up to 40% of the theoretical peak) and GPU (70–96 GB/s; 67% of the theoretical peak).

- On multiple GPUs, MPI communication over the Gigabit Ethernet network dominated the execution time and no speed-up could be observed. Additional overhead is imposed by CPU-GPU memory transfers, however for denser matrices their impact on execution time was negligible. In consequence, if Infiniband interconnect was used, the hybrid DCG solver could achieve good parallel efficiency on multiple GPUs.

- The DBDS method is dominated by solving sparse triangular systems, however it is difficult to implement it effectively on the GPU due to limited parallelisation opportunities determined by data dependencies in triangular factors.

- The mixed-precision DBDS allows us to perform the factorisation, and to solve triangular systems in single precision, while maintaining high precision of the final result. Depending on the complexity of triangular factors, on the CPU the execution time can be reduced by up to 30%. Due to the way the

triangular systems are solved on the GPU, the performance improvement from mixed-precision iterative improvement was minimal.

- The triangular solve was faster on the GPU (by a factor of 6.5–7.5 depending on precision) only for the `parabolic_fem` matrix. For all other linear systems considered in experiments, the GPU implementation was 6–14 times slower — each triangular solve step required hundreds or even thousands of CUDA kernel calls, effectively limiting the utilisation of the GPU. In consequence, in most cases it is impractical to run the DBDS solver on the GPU.

The DBDS method did not perform well on hybrid parallel platforms, however some positive results could be observed on a multi-CPU cluster.

- Subdomain overlapping can significantly improve the convergence rate of the DBDS method. The trade-off is that local linear systems are larger, and require more time for factorisation and triangular solve steps. In consequence, the impact of overlapping on time-effectiveness of the DBDS method strongly depends on the sparsity pattern of a matrix. Indeed, for the 2D Poisson matrix the time-effectiveness improves with the increasing level of overlap, but in the case of the 3D Poisson's Equation the effect is limited.

- On a large CPU cluster with Infiniband interconnect, the parallel efficiency of the DCG method is limited by the need to subtract old and new global solution vectors in each iteration. When run for a fixed number of iterations, the parallel efficiency of the non-overlapped DBDS method is significantly better, however the impact of the communication overhead becomes noticeable when the number of cores is large (64, 128). The convergence rate of both methods deteriorates with an increasing number of cores (subdomains). In consequence, the performance and scalability of DCG and non-overlapped DBDS is poor.

- However, when the level of overlap is optimised for a particular linear system, the DBDS method can perform better than state-of-the-art algorithms. In the case of 2D Poisson's Equation on $2{,}048 \times 2{,}048$ grid, the optimised DBDS method on eight CPU cores is 13.5 times faster than the PCG method, and 5.4 times faster than the fastest direct sparse solver.

Part III
Parallel Fast Poisson Solvers

The last part of this thesis is focused on Fast Poisson Solvers (FPS) based on Fourier analysis. Unlike iterative improvement algorithms considered in the previous chapters, FPS methods are *direct* and are among the fastest solvers for Poisson's Equation. FPS methods are based on the Discrete Sine Transform (DST) and the solution of tridiagonal systems — both operations are susceptible to efficient parallelisation on advanced computing platforms.

Chapter 9 explains the theory behind the FPS solver for 2D Poisson's Equation and introduces an implementation optimised for distributed memory systems — our solver (DFPS) allows for a significant reduction of the communication overhead, and in consequence, improves the scalability on multi-CPU and multi-GPU installations.

A trade-off for good scalability of DFPS on platforms with numerous processing elements, is the requirement to use a suboptimal algorithm to compute the DST. At a cost of a reduction in the maximum number of processing elements (typically several CPU cores, or up to four GPUs), the DST operation can be computed with the Fast Fourier Transform, which is an order of magnitude faster. Chapter 10 presents the modified solver, and numerical experiments on the 2D Poisson's Equation that confirm an outstanding performance achieved by solvers following this approach.

The FFT-based implementation of the FPS method can be adjusted to accommodate the 3D Poisson's Equation. In this case, a multi-dimensional DST has to be performed. Since it involves completely different memory access patterns, a number of optimisations are required to ensure good performance. The FPS solver for the 3D Poisson's Equation is presented in Chapter 11.

Fast Poisson Solver implementations for GPU platforms have been published in the literature already, however they are typically designed to run on a single graphics card and the maximum problem size is limited by the relatively small GPU memory. Chapters 10 and 11 introduce an innovative *streaming processing* method that addresses both issues — our streaming FPS is the first solver to handle arbitrarily large Poisson grids (even larger than the GPU memory) and to enable multi-GPU processing with linear speed-up.

Chapter 9

Distributed Fast Poisson Solver

Fast Poisson Solvers (FPS) based on Fourier analysis are among the fastest methods used to solve Poisson's Equation. As will be shown in this and the following two chapters, these algorithms offer performance close to optimal in terms of time complexity analysis. Furthermore, the formulation of the FPS approach allows for efficient parallelisation on advanced architectures.

This chapter introduces the FPS method and presents the distributed memory implementation that can run on traditional multi-CPU clusters, as well as on hybrid CPU-GPU systems. The following Chapters 10 and 11 describe shared memory implementations for 2D and 3D Poisson's Equation, respectively. The latter approach trades scalability for an order of magnitude faster operations.

9.1 Introduction

The application of Fourier analysis to solve Poisson's Equation has been considered for almost 50 years — the pioneering work was published by Hockney (1965). These results were then generalised for a broader range of partial differential equations (Pickering, 1986; Duffy, 2010).

The FPS formulation sketched by Kincaid and Cheney (2002, §9.9) became a starting point for the 2D Poisson's Equation solver presented in this chapter. The solver is based on the observation that the initial problem can be transformed to a set of

tridiagonal systems, which can be solved independently. The Discrete Sine Transform (DST) is used for the purpose of problem transformation and is derived in the following section. An analysis of this approach is also provided by Demmel (1996).

The Fast Poisson Solver benefits from a number of sources of parallelism. DST is a special case of Discrete Fourier Transform and can be efficiently calculated using the Fast Fourier Transform (FFT) that can be parallelised. A range of high-performance FFT implementations are available, e.g. the FFTW3 library (Frigo and Johnson, 2005), which is designed for multi-core CPUs, and the CUFFT library (NVIDIA, 2012b; available as a part of the CUDA Toolkit) utilising numerous GPU cores.

Finding solutions to multiple tridiagonal systems can benefit from two sources of parallelism: each tridiagonal system can be solved independently, and a parallel tridiagonal solver can be used. The Thomas Algorithm is inherently sequential (Atkinson and Han, 2004, §6.4), but it is simple to implement, has the optimal time complexity, and so makes a good choice on the CPU. Among parallel tridiagonal solvers are Cyclic Reduction, Parallel Cyclic Reduction, and Recursive Doubling. These methods can provide good performance on parallel architectures, however communication overheads make efficient implementation difficult. A number of tridiagonal solvers on the GPU were described and compared by Zhang et al. (2010) and Göddeke and Strzodka (2011), but their implementations were limited by a relatively small GPU shared memory size. To overcome that issue, a range of auto-tuned solvers were proposed by Davidson et al. (2011) and Kim et al. (2011).

Assuming a square, structured grid of $n \times n$ points, the time complexity of the DST operation is $\mathcal{O}(n^2 \log n)$, and the time required to solve n tridiagonal systems is $\mathcal{O}(n^2)$, therefore the time complexity of the sequential Fast Poisson Solver is $\mathcal{O}(n^2 \log n)$. This compares favourably to PCG, which requires $\mathcal{O}(n^3)$ time, but is not as fast as the sequential Multigrid Method that can solve the Poisson problem in $\mathcal{O}(n^2)$ time, which is optimal order-wise (Demmel, 1997, p. 277).

The Multigrid Method is difficult to parallelise and fails to deliver optimal performance in the theoretical PRAM model (Parallel Random-Access Machine, Keller et al., 2001). The Fast Poisson Solver, which is not optimal on sequential platforms, has the *optimal* complexity on parallel architectures. However, the shortcoming of the PRAM model is that it does not take data transfers into account — the cost of

communication can easily become the main factor limiting the performance of any parallel algorithm. Therefore, several implementations, designed for distributed and shared memory systems, were considered in this study.

In this chapter, only distributed memory systems were considered. The simplest approach, which simply divides work in each phase among all processing elements (PEs), requires two expensive collective communication operations — `Allgather`. The cost of this operation varies depending on the actual implementation and network topology. In the optimal case, the time complexity is $\mathcal{O}(\log P + n)$, but in some cases the execution time increases linearly with the number of PEs (Bruck et al., 1997). In this case, `Allgather` can quickly dominate the total execution time. In this study, an alternative approach was considered. By performing the partial, instead of the full DST, only one `Allgather` operation is needed.

However, the partial DST cannot be computed as efficiently as the full DST. A possible solution is to use the algorithm proposed by Goertzel (1958) for separate DST frequencies, but in our case the contiguous range of frequencies is calculated on each PE, therefore the partial DST can be expressed in terms of the dense matrix-matrix multiplication (GEMM operation in BLAS, Basic Linear Algebra Subprograms; Blackford et al., 2002). The latter approach has the same time complexity, however it can benefit from several highly optimised implementations of GEMM.

9.2 Fast Poisson Solver

This section describes the Fast Poisson Solver for distributed memory systems (DFPS). First, in Subsection 9.2.1 mathematical foundations of the method are presented. Then, various versions of the algorithm are outlined and parallelisation opportunities are discussed in Subsection 9.2.2.

9.2.1 Method Derivation

The general form of the 2D Poisson's Equation is defined as

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f \tag{9.1}$$

on some domain Ω. In this study, rectangular domains are considered, i.e.

$$\Omega = \{(x, y) : x_{min} < x < x_{max},\ y_{min} < y < y_{max}\}. \tag{9.2}$$

Using $n \times m$ grid points uniformly distributed in each dimension and using the FDM method, Equation (9.1) on domain Ω can be discretised to a set of equations

$$h_x^{-2}\left(v_{i-1,j} - 2v_{i,j} + v_{i+1,j}\right) + h_y^{-2}\left(v_{i,j-1} - 2v_{i,j} + v_{i,j+1}\right) = f_{i,j} \tag{9.3}$$

$$\text{assuming: } h_x = \frac{x_{max} - x_{min}}{n+1},\ h_y = \frac{y_{max} - y_{min}}{m+1},\ v_{i,j} \approx \phi(x_i, y_j),\ f_{i,j} = f(x_i, y_j),$$

$$\text{where } x_i = x_{min} + ih_x\ (1 \le i \le n), \quad \text{and } y_j = y_{min} + jh_y\ (1 \le j \le m).$$

The Discrete Sine Transform comes in several flavours, but in this study DST-I is considered (Press et al., 2007, §12.4.1). The transform is defined as follows:

$$v_{i,j} = \sum_{k=1}^{n} \hat{v}_{k,j} \sin(ik\varphi), \tag{9.4}$$

$$f_{i,j} = \sum_{k=1}^{n} \hat{f}_{k,j} \sin(ik\varphi), \quad \text{where } \varphi = \frac{\pi}{n+1}. \tag{9.5}$$

When $v_{i,j}$ values are substituted according to Equation (9.4), Equations (9.3) can be rewritten as

$$h_x^{-2} \sum_{k=1}^{n} \hat{v}_{k,j} \left[\sin((i-1)k\varphi) - 2\sin(ik\varphi) + \sin((i+1)k\varphi)\right] + \\ + h_y^{-2} \sum_{k=1}^{n} \left(\hat{v}_{k,j-1} - 2\hat{v}_{k,j} + \hat{v}_{k,j+1}\right) \sin(ik\varphi) = f_{i,j}. \tag{9.6}$$

The expression in the square brackets in Equations (9.6) can be simplified with the following trigonometric identity (substituting $A = ik\varphi$ and $B = k\varphi$):

$$\begin{aligned} &\sin(A+B) - 2\sin A + \sin(A-B) = \\ &\quad = -2\sin A + 2\sin A \cos B && \left(\sin\alpha + \sin\beta = 2\sin\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}\right) \\ &\quad = -4\sin A\left(\frac{1-\cos B}{2}\right) \\ &\quad = -4\sin A \sin^2\frac{B}{2} && \left(\sin^2\frac{\alpha}{2} = \frac{1-\cos\alpha}{2}\right). \end{aligned} \tag{9.7}$$

This allows us to rewrite Equations (9.6) in the form

$$h_x^{-2}\sum_{k=1}^{n} -4\hat{v}_{k,j}\left(\sin^2\frac{k\varphi}{2}\right)\sin(ik\varphi)+ \\ + h_y^{-2}\sum_{k=1}^{n}\left(\hat{v}_{k,j-1}-2\hat{v}_{k,j}+\hat{v}_{k,j+1}\right)\sin\left(ik\varphi\right)=\sum_{k=1}^{n}\hat{f}_{k,j}\sin\left(ik\varphi\right). \quad (9.8)$$

A matrix of elements of the form $\sin(ij\varphi)$ is non-singular (Kincaid and Cheney, 2002, §9.9), thus the problem can be reduced to the solution of the following equations:

$$h_y^{-2}\hat{v}_{k,j-1}-\left(2h_y^{-2}+4h_x^{-2}\sin^2\frac{k\varphi}{2}\right)\hat{v}_{k,j}+h_y^{-2}\hat{v}_{k,j+1}=\hat{f}_{k,j}. \quad (9.9)$$

For a fixed k, this becomes a tridiagonal system with m unknowns. Since $1 \leq k \leq n$, there are n such systems to be solved.

To calculate the $\hat{f}_{i,j}$ values, the inverse DST-I transformation has to be performed. In this case, it is the DST-I operation, multiplied by a scalar factor, namely

$$\hat{f}_{i,j}=\frac{2}{n+1}\sum_{k=1}^{n} f_{k,j}\sin(ik\varphi).$$

The DST-I transformation can be expressed in terms of matrix-matrix multiplication. Equation (9.4) can be rewritten as

$$v_{i,j}=\begin{bmatrix}\sin(i\varphi) & \sin(i\cdot 2\varphi) & \dots & \sin(in\varphi)\end{bmatrix}\cdot\begin{bmatrix}\hat{v}_{1,j} & \hat{v}_{2,j} & \dots & \hat{v}_{n,j}\end{bmatrix}^T, \quad (9.10)$$

and the whole transformation can be expressed as

$$\mathbf{V}=\begin{bmatrix} v_{1,1} & v_{1,2} & \dots & v_{1,m}\\ v_{2,1} & v_{2,2} & \dots & v_{2,m}\\ \vdots & \vdots & \ddots & \vdots\\ v_{n,1} & v_{n,2} & \dots & v_{n,m}\end{bmatrix}= \quad (9.11)$$

$$=\begin{bmatrix} \sin(\varphi) & \sin(2\varphi) & \dots & \sin(n\varphi)\\ \sin(2\varphi) & \sin(2\cdot 2\varphi) & \dots & \sin(2\cdot n\varphi)\\ \vdots & \vdots & \ddots & \vdots\\ \sin(n\cdot\varphi) & \sin(n\cdot 2\varphi) & \dots & \sin(n\cdot n\varphi)\end{bmatrix}\cdot\begin{bmatrix}\hat{v}_{1,1} & \hat{v}_{1,2} & \dots & \hat{v}_{1,m}\\ \hat{v}_{2,1} & \hat{v}_{2,2} & \dots & \hat{v}_{2,m}\\ \vdots & \vdots & \ddots & \vdots\\ \hat{v}_{n,1} & \hat{v}_{n,2} & \dots & \hat{v}_{n,m}\end{bmatrix}=\mathbf{T}\hat{\mathbf{V}},$$

where $\mathbf{V} = [v_{i,j}]$, $\hat{\mathbf{V}} = [\hat{v}_{i,j}]$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$, and $\mathbf{T}$ is an n-by-n transformation matrix. The matrix for the inverse DST-I can be easily calculated:

$$\mathbf{T}^{-1} = \frac{2}{n+1}\,\mathbf{T}\,. \tag{9.12}$$

9.2.2 Parallel Fast Poisson Solvers

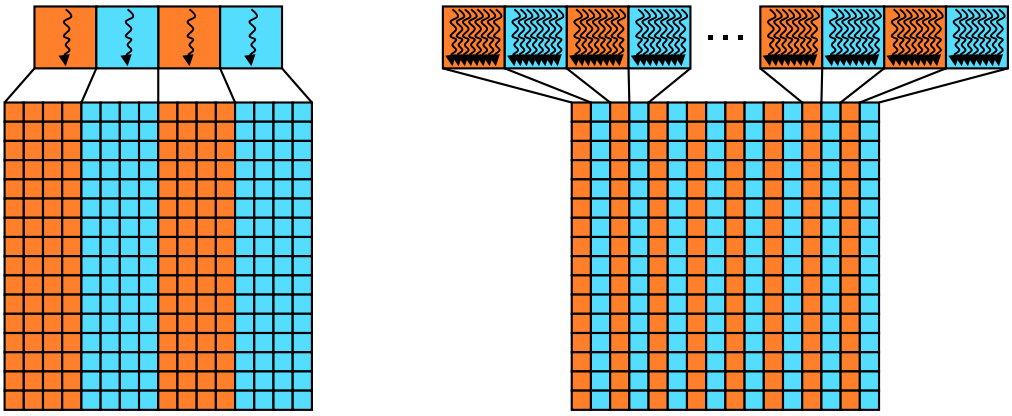
The sequential Fast Poisson Solver derived from the equations in the previous subsection is outlined in Algorithm 9.1. All three steps can benefit from two levels of parallelism. The first one is straightforward, as each processing element can be assigned a separate chunk of data to work on with sequential algorithms: transforming separate data vectors in DST, or solving different tridiagonal systems (Figure 9.1 A). This coarse-grain parallelism is usually enough for multi-core CPU clusters, since the number of processing elements is typically small compared to the problem size. Furthermore, in the presence of shared memory there are no communication costs associated with coarse parallelism, typically offering almost linear speed-ups.

Algorithm 9.1 Sequential Fast Poisson Solver

1: Calculate $\hat{f}_{i,j}$ values { DST-I on m vectors of size n }
2: Calculate $\hat{v}_{i,j}$ values { Solve n tridiagonal systems of size $m \times m$ }
3: Calculate $v_{i,j}$ values { DST-I on m vectors of size n }

Both operations can also be parallelised at the finer level — multiple processing elements can work on the same data chunk (Figure 9.1 B). The prospective further speed-up comes at the price of more complicated code, and communication and synchronisation overheads. To obtain high performance, this approach typically requires shared memory and is covered in the next two chapters.

The straightforward Fast Poisson Solver with coarse-grain parallelisation in a distributed memory system is presented in Figure 9.2. This approach requires three collective communication operations: one `Gather` (gathers distributed data on one node), and two `Allgather` operations (same as `Gather`, but also distributes the result to all the nodes). The total amount of data sent in the `Gather` operation is proportional to the grid size — $\mathcal{O}(n^2)$, assuming $n = m$ — and scales well with increasing number of processing elements. However, the total amount of transferred



(A) Coarse-grain parallelism (B) Fine-grain parallelism

FIGURE 9.1: Comparison of coarse- and fine-grain parallelism in the Fast Poisson Solver.

data in the `Allgather` operation grows linearly with the number of processing elements P, resulting in $\mathcal{O}(n^2 P)$ complexity. In the DFPS solver, this quickly becomes the dominating factor, i.e. when $P > \mathcal{O}(\log n)$, and greatly limits the scalability.

If the full DST is replaced with the *partial* DST, one of the `Allgather` operations can be eliminated. In this case, there is no need to exchange the data between DST and tridiagonal solve phases, as they now operate on the same data chunk. The modified method is presented in Algorithm 9.2. This modification increases the work complexity of the method to $\mathcal{O}(n^3)$, but allows for better scalability — now $\mathcal{O}(\sqrt{n})$ processing elements can be efficiently utilised.

ALGORITHM 9.2 Distributed Fast Poisson Solver (DFPS)

1: Calculate local $\hat{f}_{i,j}$ values { The partial DST on m vectors of size $\frac{n}{P}$ }
2: Calculate local $\hat{v}_{i,j}$ values { Solve $\frac{n}{P}$ tridiagonal systems of size $m \times m$ }
3: Gather local $\hat{v}_{i,j}$ values and distribute to all nodes { The `Allgather` operation }
4: Calculate local $v_{i,j}$ values { The partial DST on m vectors of size $\frac{n}{P}$ }
5: Gather local $v_{i,j}$ values { The `Gather` operation }

To show that, let us consider an $n \times n$ grid. If n is sufficiently large, the solution time as a function of the number of processing elements P can be expressed as

$$T(P) = 2T_{DST}(P) + T_{Allgather} = 2\,\frac{2n^3}{P \cdot F_{DST}} + (P-1)\,\frac{n^2}{F_{Allgather}}. \tag{9.13}$$

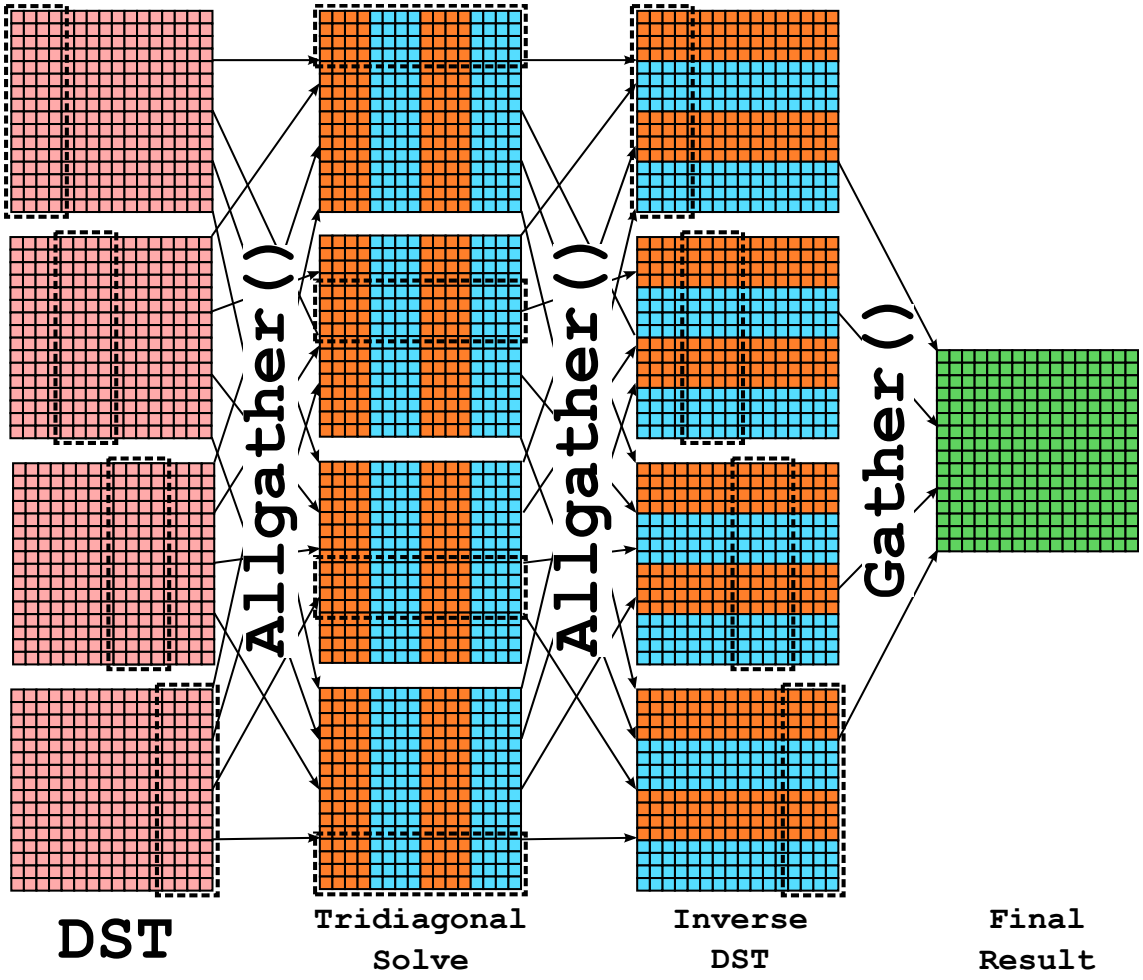


FIGURE 9.2: The Distributed Fast Poisson Solver on four processing elements (two `Allgather` operations). The dotted frames mark the data chunks processed at the respective stage. Orange and blue colours indicate data chunks received from different processing elements in the `Allgather` operation.

Here, F_{DST} denotes the instruction throughput in the DST step on each processing element and $F_{Allgather}$ denotes the bandwidth of the underlying network. To find the minimum of $T(P)$ it is necessary to calculate the first derivative

$$\begin{aligned} T'(P) &= -P^{-2}\frac{4n^3}{F_{DST}} + \frac{n^2}{F_{Allgather}} = 0 \\ P^2 &= \frac{4nF_{Allgather}}{F_{DST}} \\ P &= C\sqrt{n}. \end{aligned} \tag{9.14}$$

Therefore, the optimal number of processing elements $P = \mathcal{O}(\sqrt{n})$.

9.3 Implementation Details

In this study, two DFPS implementations were compared. The first was designed for large distributed memory CPU clusters and used MPI for communication. The second was designed for distributed GPU clusters, i.e. systems where multiple GPUs are connected to separate motherboards.

Subsection 9.3.1 presents the implementation details of each algorithm step. Then, Subsection 9.3.2 discusses how GPUs can be used in the hybrid CPU-GPU implementation. Finally, Subsection 9.3.3 lists libraries that were used in the development.

9.3.1 Algorithm Steps

The partial DST and its inverse were calculated following Equation (9.11). If the transformation matrix $\mathbf{T}$ is multiplied by $\sqrt{\frac{2}{n+1}}$, it can be used for both forward and inverse DST. The rows of matrix $\mathbf{T}$ assigned to each processing element are pre-calculated and stored in memory.

The tridiagonal solve step is an order of magnitude faster than the DST transformation and quickly becomes negligible in the total execution time, therefore only coarse-grain parallelism was considered — each system was solved with the Thomas Algorithm, which offers the optimal time complexity on sequential machines. The pseudo-code is presented in Algorithm 9.3.

Algorithm 9.3 The Thomas Algorithm for solving tridiagonal systems

```
// In the beginning, x contains the right-hand side vector.
// a, b, and c are lower, main, and upper diagonals, respectively.
// aux is an auxiliary array of size n.
template <typename T>
void Thomas(size_t n, const T* a, const T* b, const T* c, T* x, T* aux)
  aux[0] = c[0] / b[0];
  x[0] = x[0] / b[0];
  for (size_t i = 1; i < n; i++) {
    aux[i] = c[i] / (b[i] - a[i] * aux[i-1]);
    x[i] = (x[i] - a[i] * x[i-1]) / (b[i] - a[i] * aux[i-1]);
  }

  for (size_t i = n-1; i--- > 0; ) {
    x[i] := x[i] - aux[i] * x[i+1];
  }
}  // At the end, x contains the solution vector.
```

The FPS solver requires three memory buffers, each containing $\frac{n^2}{P}$ elements. The first one is used to store the transformation matrix, the second one is for the intermediate result of DST and inverse DST before `Allgather` and `Gather` operations, and the third one is an auxiliary memory for the Thomas Algorithm. If the original source term input data is to be preserved, an additional buffer of n^2 elements is required.

9.3.2 Hybrid Implementation

The hybrid implementation follows the same design as the CPU-only implementation, but performs DST, inverse DST, and tridiagonal solve operations on the GPU. The buffer holding transformation matrix elements resides in global memory and is pre-computed on the GPU, rather than on the CPU. In addition, a buffer of n^2 elements is kept in global memory for the source term before the DST, and for $\hat{v}_{i,j}$ elements before the inverse DST, and two buffers of $\frac{n^2}{P}$ elements for the forward, and inverse DST results, and as an auxiliary memory for the tridiagonal solver.

Similar to the CPU implementation, the tridiagonal solve step takes only a small fraction of the total computation time. Therefore, it was decided to use a GPU

implementation of the Thomas Algorithm — one thread solves one tridiagonal system. Since each system can be solved independently, this approach still benefits from parallel capabilities of the GPU. Tridiagonal solvers with more favourable theoretical properties, e.g. Cyclic Reduction, Parallel Cyclic Reduction or Recursive Doubling, could also be used, but they would only offer negligible improvement in overall performance at the significant development cost.

Typically, FPS operations are many times faster on the GPU, but there is an additional CPU-GPU communication cost. The hybrid implementation requires two transfers of n^2 elements to the GPU, and two transfers of $\frac{n^2}{P}$ elements from the GPU. In this case communication cannot be overlapped with computation. Implementations benefiting from such an overlap are discussed in Chapters 10 and 11.

9.3.3 Libraries

The proposed algorithm can work with any implementation of MPI and BLAS standards. In this study, only open-source libraries were considered, as they typically offer decent performance and robustness at no financial cost. Should the need arise, these can be easily replaced with commercial libraries, or even hand-crafted high-performance implementations of particular operations, e.g. `Allgather` proposed by Gupta and Vadhiyar (2007).

The initial experiments confirmed that MPICH (`http://www.mpich.org/`, last access: 1 July 2013) provides the best and the most consistent performance among the open-source MPI implementations. Similar experiments showed that ATLAS (Whaley and Petitet, 2005; `http://math-atlas.sourceforge.net/`, last access: 1 July 2013) provides the best performance of BLAS operations on the CPU. For GPUs, CUBLAS (NVIDIA, 2012a) is the most mature library and typically offers outstanding performance. As of CUDA Toolkit 5.0.35, the GEMM routine yields performance similar to the fast implementation developed by Nath et al. (2010).

9.4 Numerical Experiments

All machines used in the experiments were equipped with Intel Core i7-980x CPU (12 MB cache, 3.33 GHz, six cores) with 12 GB main memory, were running the

Ubuntu 12.04 Server (64-bit) operating system, and were connected with the Gigabit Ethernet network. The CPU code was compiled with `gcc` version 4.6.3, and the GPU code was compiled with `nvcc` release 5.0, V0.2.1221 and CUDA Toolkit 5.0.35. Technical details of two GPU models used in the experiments (Tesla C2050 and GTX 480) are summarised in Table B.2 in Appendix B. Unless specified otherwise, all experiments were run in single and double floating-point arithmetic, were repeated ten times, results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5% (less than 1% in most cases).

All results were validated for correctness using the formula for *scaled residual norm* proposed by Barrett et al. (1994, Ch. 4):

$$res = \frac{||\mathbf{r}||}{||\mathbf{A}|| \cdot ||\mathbf{x}|| + ||\mathbf{b}||}. \tag{9.15}$$

All res values were confirmed to be lower than 2ε, where ε is the floating-point representation error. On a machine with IEEE standard floating-point arithmetic, $\varepsilon = 2^{-24} \approx 10^{-7}$ in single precision, and $\varepsilon = 2^{-53} \approx 10^{-16}$ in double precision.

The experiments were conducted on the following model 2D Poisson problem with homogeneous Dirichlet boundary conditions (the same as in Chapter 4):

$$\begin{aligned} \Omega &= \{(x, y) : -1 < x,\ y < 1\}, \\ \nabla^2 \phi &= -2\pi^2 \sin(\pi x) \sin(\pi y) \text{ in } \Omega, \\ \phi &= 0 \text{ on } \partial\Omega. \end{aligned}$$

The solution to this problem is presented in Figure C.1 A in Appendix C.

Subsection 9.4.1 presents models used to evaluate the performance of the DFPS solver. Then, the following subsections present results of experiments on: small-scale CPU-based parallel platforms (Subsection 9.4.2), small distributed GPU cluster (Subsection 9.4.3), and the CPU-based supercomputer Astral (Subsection 9.4.4). Finally, Subsection 9.4.5 contains estimates of the DFPS performance on the distributed GPU cluster with Infiniband interconnect.

9.4.1 Performance Models

To ensure the reliable performance measurement and fair comparison between algorithms, the following performance models were used:

$$F_{dst}(n,m) = \frac{nm \cdot (2n-1)}{t_{dst} \cdot 10^9} \quad [GFLOPS] \tag{9.16}$$

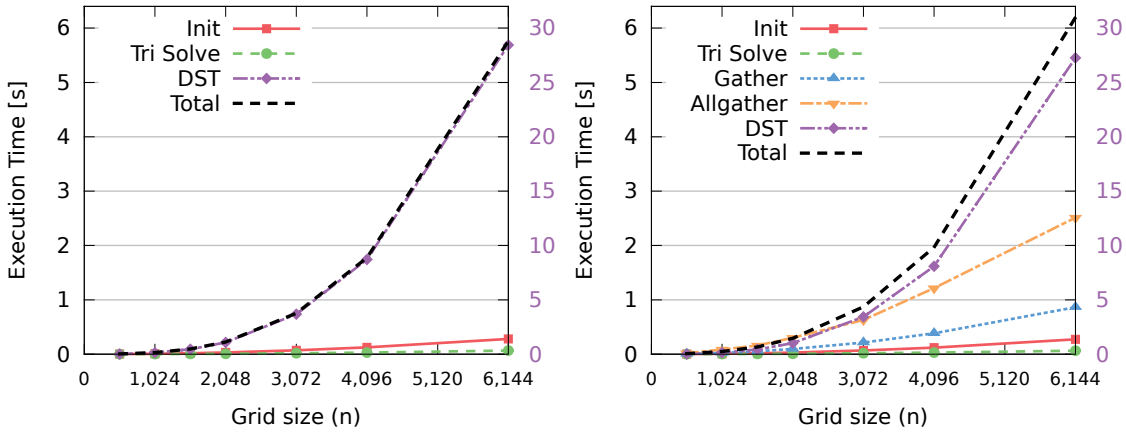
$$F_{tri}(n,m) = \frac{8nm}{t_{tri} \cdot 10^9} \quad [GFLOPS] \tag{9.17}$$

$$F_{overall}(n,m) = \frac{2nm \cdot (2n+3)}{t_{overall} \cdot 10^9} \quad [GFLOPS] \tag{9.18}$$

Here, n and m denote the number of rows and columns in the grid, respectively. In all the experiments only square grids were considered, i.e. $n = m$.

9.4.2 Small-Scale CPU-based Parallel Platforms

The first step was to assess the performance of the Fast Poisson Solver on small-scale parallel platforms: single multi-core CPU (shared memory) and a small cluster of multi-core CPUs connected with the Gigabit Ethernet network.



(A) Shared memory (B) Distributed memory

FIGURE 9.3: Execution time of each phase of DFPS on six cores/CPUs (single precision). The right scale is for the dominating DST step and total execution times.

The DFPS solver consists of the following steps: initialisation, DST, tridiagonal solve, partial solutions exchange (an `Allgather` operation), inverse DST, and gathering the final solution (a `Gather` operation). Communication steps are not required in the presence of shared memory (multi-core CPU).

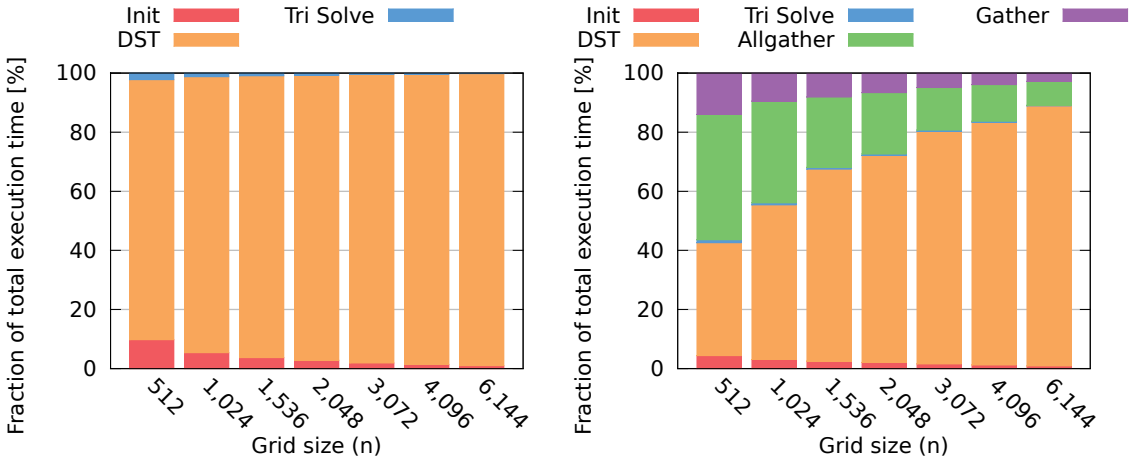
Calculating the DST and its inverse were dominating in the total execution time in shared (Figure 9.3 A) and distributed (Figure 9.3 B) memory systems. Other computation steps quickly became negligible, which is consistent with the fact that the time complexity of the partial DST is $\mathcal{O}(\frac{n^3}{P})$, compared to $\mathcal{O}(\frac{n^2}{P})$ for other operations. The `Allgather` operation required $\mathcal{O}(n^2 P)$ time and even on six distributed cores had a noticeable impact on the total execution time.

Execution times reported in Figure 9.3 were measured for single precision arithmetic. When double precision is used instead, the execution time for initialisation and partial DST increased by roughly 10%, for the tridiagonal solve step by roughly 30%, and for communication operations was doubled.

The detailed execution time breakdown confirmed that the partial DST is a dominant operation (Figure 9.4), but also showed that communication can take a significant fraction of time for smaller problems. This indicates that the scalability of DFPS is limited — up to $\mathcal{O}(\sqrt{n})$ processing elements can be efficiently utilised (cf. Subsection 9.2.2). In order to improve the scalability, a faster interconnect between the processing elements is required, e.g. Infiniband.

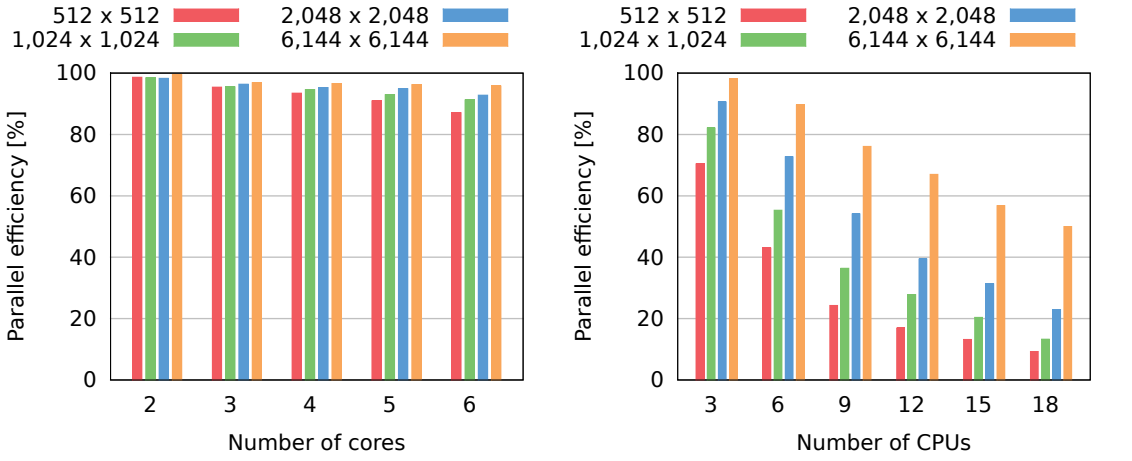
In the presence of shared memory, the Fast Poisson Solver delivered close to linear speed-up (Figure 9.5 A) since there are no communication overheads. However, the efficiency decreased slightly with increasing number of cores used, especially on small grids. This was connected with the typical concurrency overheads: scheduling, and the fact that all the cores share the same CPU cache. As the computation time became shorter for small grids, the impact of these overheads was more prominent.

In contrast to the shared memory system, parallel efficiency of DFPS run on the distributed memory system decreased more rapidly with the increasing number of cores (Figure 9.5 B). The increasing fraction of time spent on the `Allgather` operation was responsible for this trend. Nevertheless, the solver could deliver decent parallel efficiency of 50% for large grids. As expected, when the number of cores was fixed, then the parallel efficiency increased significantly with the problem size.



(A) Shared memory (B) Distributed memory

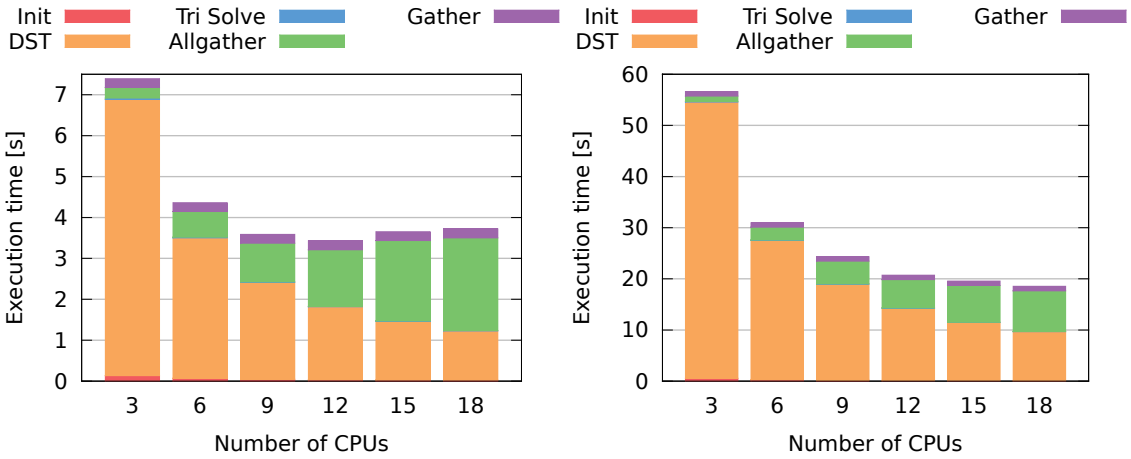
FIGURE 9.4: DFPS execution time breakdown in two memory models (single precision). DST is a dominant operation, however in distributed memory systems the cost of communication may become significant.



(A) Shared memory (B) Distributed memory

FIGURE 9.5: Parallel efficiency of DFPS in two memory models (single precision). The efficiency in the distributed memory system decreases noticeably due to the increasing communication overhead.

Figure 9.6 shows how the fraction of time spent on communication with the increasing number of processing elements P on large grids. In the distributed system, the constant in Equation (9.14) $C \approx 0.23$ and indicates that the optimal $P = 12$ for $n = 3{,}072$, and $P = 18$ for $n = 6{,}144$. Indeed, the fastest solution time could be observed on 12 cores (Figure 9.6A), and the times spent on `Allgather` and DST operations are roughly the same. Similar results were observed for a $6{,}144 \times 6{,}144$ grid and 18 cores (Figure 9.6B).



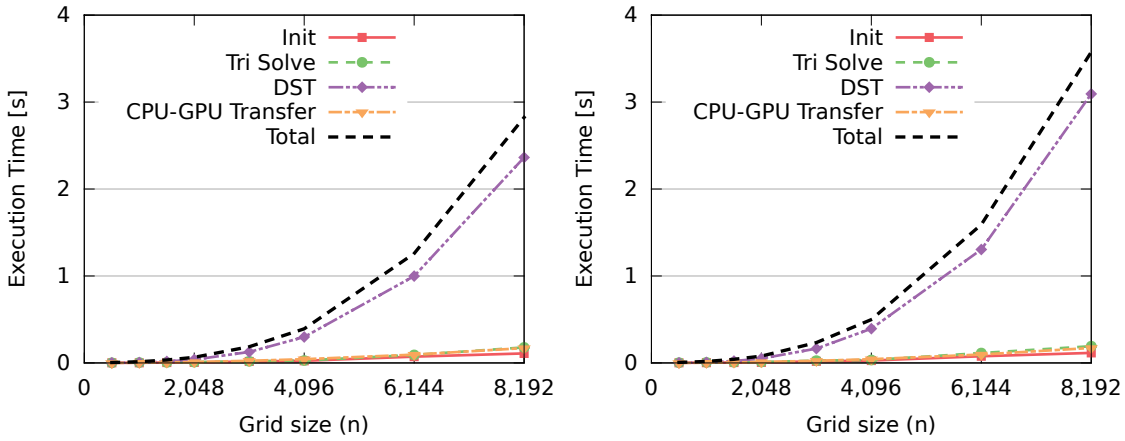
(A) Grid size: $3{,}072 \times 3{,}072$ (B) Grid size: $6{,}144 \times 6{,}144$

FIGURE 9.6: DFPS execution time breakdown in the distributed memory system (single precision). For a fixed grid size communication overhead connected with the `Allgather` operation increases significantly with the number of cores. The optimal number of processing elements scales in the $\mathcal{O}(\sqrt{n})$ fashion — in this case, times spent on computation and communication are almost equal.

Regardless of the grid size and the number of processing elements, each step was executed at a consistently high instruction throughput rate. The LINPACK benchmark (Intel, 2012) indicated the peak performance of roughly 60 GFLOPS. The matrix-matrix multiplication (the DST step) was performed at roughly 33 GFLOPS (30 GFLOPS in double precision), which is more than 50% of the peak performance. The tridiagonal solver ran consistently at 4.5 GFLOPS (3.25 GFLOPS in double precision), but this operation has low computation-to-communication ratio and is inherently memory-bound. Both communication steps utilised 900–930 Gbps out of the theoretical 1000 Gbps in the Gigabit Ethernet network.

9.4.3 Small Distributed GPU Cluster

Both graphics cards used in experiments delivered an outstanding performance (Figure 9.7). The solution of 2D Poisson's Equation on a $6{,}144 \times 6{,}144$ grid on a six-core CPU required roughly 30 seconds, compared to just over a second on GTX 480 (Figure 9.7A), and 1.5 seconds on Tesla C2050 (Figure 9.7B).



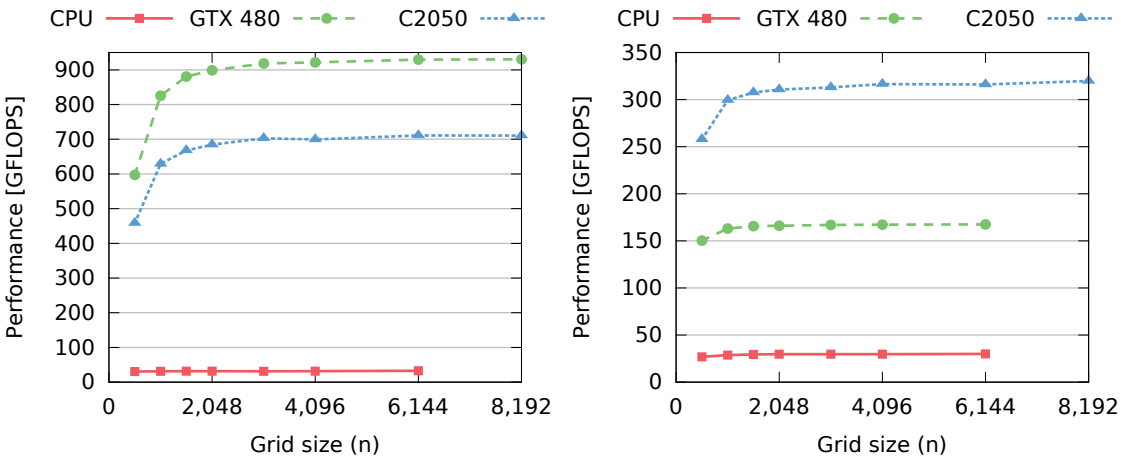
(A) GeForce GTX 480 (B) Tesla C2050

FIGURE 9.7: Execution time of each step of DFPS on a single GPU (single precision).

Similar to the CPU implementation, calculating matrix-matrix products was the dominant operation and determined the overall performance — all other operations: initialisation, tridiagonal solve, and CPU-GPU transfers, quickly became negligible in the total execution time. This justifies the decision to use the Thomas Algorithm to solve tridiagonal systems, rather than more complex methods designed for parallel architectures — the potential improvement of the overall performance is minimal.

Figure 9.8 presents the instruction throughput observed for DST on the CPU and two different GPUs. Unlike the CPU implementation, GPUs reached the peak performance only for sufficiently large grids: at least $n \geq 2{,}048$ in single precision and $n \geq 1{,}024$ in double precision. In single precision, both GPUs obtained roughly 67% of their theoretical peak performance — more than 900 GFLOPS on GTX 480, which is roughly 30 times faster than a six-core Intel i7-980x CPU (Figure 9.8A).

An interesting result can be observed for double precision arithmetic. As expected, Tesla C2050 delivered roughly half of the single precision throughput, however the



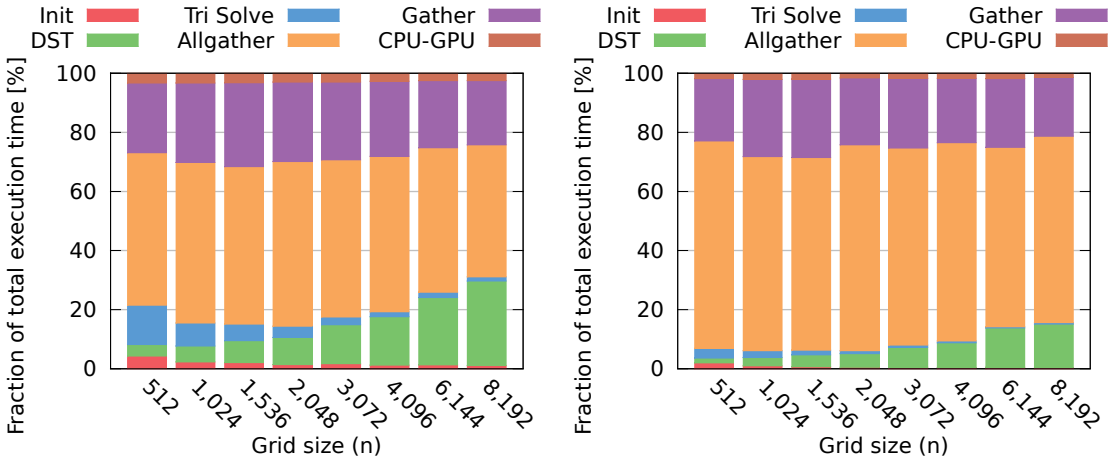
(A) Single precision (B) Double precision

FIGURE 9.8: The DST performance on CPU (six cores) and two GPUs. In single precision, both GPUs run at 67% of their theoretical peak performance, up to 30 times faster than a high-end CPU. GTX 480 is designed for computer games and is not optimised for computation in double precision. In this case, Tesla C2050 is faster.

performance on GTX 480 was significantly lower (Figure 9.8B). Both graphics cards are based on the same architecture which is capable of fast double precision operations. However, NVIDIA decided to limit the double precision performance on gaming cards (e.g. GTX 480), which explains the discrepancy. Nevertheless, the double precision GPU solver ran more than 5 times (GTX 480), and almost 10 times (Tesla C2050) faster than the CPU implementation.

Significantly faster computation on graphics cards resulted in a much higher fraction of time spent on communication when using multiple GPUs (Figure 9.9). Even on just two GPUs, `Gather` and `Allgather` operations accounted for more than 60% of the total execution time (Figure 9.9A). This issue became more prominent on three GPUs — over 75% of the time was spent on MPI communication (Figure 9.9B).

In fact, total execution times observed on two and three GPUs were *higher* than on a single GPU. This difference became smaller as the problem size got larger, and indeed, the hybrid solver scales in the same $\mathcal{O}(\sqrt{n})$ fashion as the CPU implementation. However, the C constant in Equation (9.14) was much lower in this case: $C \approx 0.014$. Even for an $8{,}192 \times 8{,}192$ grid one processing element is optimal. This



(A) 2 GPUs (B) 3 GPUs

FIGURE 9.9: Execution time breakdown in hybrid DFPS run on multiple Tesla C2050 GPUs (single precision). The MPI communication dominates the execution time, effectively limiting the scalability of the solver when the relatively slow Gigabit Ethernet is used.

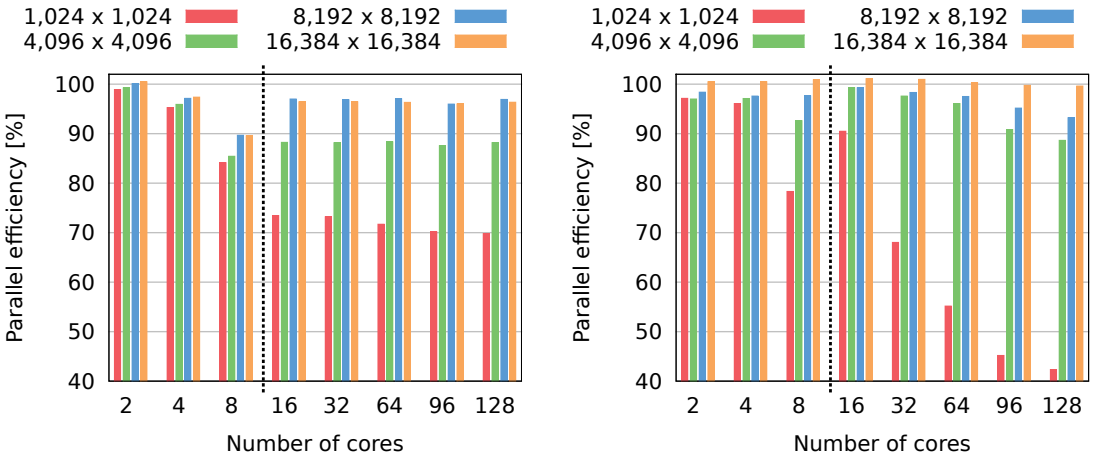
means that the solver run on multiple GPUs would become faster only for problems that are too large to fit in the GPU memory (typically $n > 13{,}000$).

The time spent on communication between the CPU and the GPU accounted for 2–3% of the total execution time. CPU-GPU data transfers were performed via a fast PCI-Express channel. Rates of 6 GB/s and 6.6 GB/s were consistently observed for data copies to- and from the GPU, respectively. As the problem size increased, the fraction of time spent on CPU-GPU communication decreased. This suggests that scalability can be improved when a faster interconnect is available, e.g. Infiniband.

9.4.4 Large CPU Cluster with Infiniband

The two previous subsections identified MPI communication as the main factor limiting DFPS performance. To confirm that using a faster interconnect can lead to a significantly better performance, experiments were carried out on the Cranfield University supercomputer — Astral. It is equipped with 80 computing nodes, each consisting of two Intel Xeon E5-2660 CPUs (20 MB cache, 2.20 GHz, 8 cores) and 64 GB of memory, i.e. each node consists of 16 cores and 4 GB of memory per core. The nodes are connected with Infiniband low-latency network (Shanley et al., 2003).

The code was compiled with Intel compiler version 12.1.0. Intel MPI Library version 4.0 update 3 was used as a Message Passing Interface implementation. The ATLAS library was not available on Astral, therefore a slower Netlib BLAS implementation (Blackford et al., 2002) was used instead.



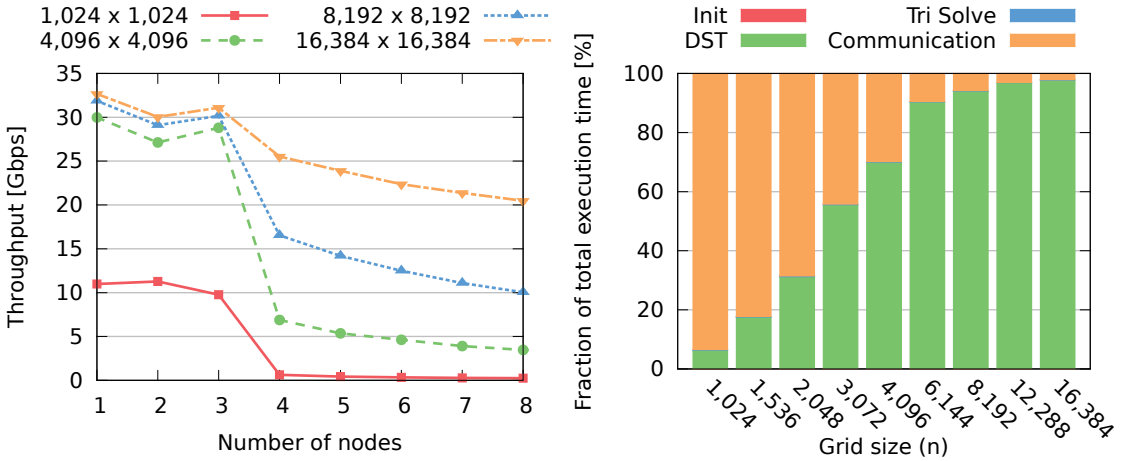
(A) Discrete Sine Transform (B) Tridiagonal Solver

Figure 9.10: Parallel efficiency of DFPS computation steps on Astral (single precision). The initial drop in performance is caused by the decreasing amount of CPU cache per core. Above the 8-core threshold, the speed-ups are close to linear for sufficiently large grids.

The DST and tridiagonal solve steps scale in almost linear fashion (parallel efficiency above 95%) for sufficiently large grids even when run on 128 cores (Figure 9.10). The initial decrease in the parallel performance could be observed for the Discrete Sine Transform (Figure 9.10A). Up to 8 cores, the computation ran on a single Astral CPU, thus the total amount of cache was constant — the cache miss ratio increased with the number of active threads. Above the 8-core threshold, the CPU cache size increased proportionally to the number of cores. In consequence, the parallel efficiency was constant for $P \leq 16$, and depended only on the grid size.

The decrease in the parallel efficiency of the tridiagonal solve step was less prominent (Figure 9.10B). Due to the row-major storage, for large grids cache is used efficiently regardless of the number of cores involved in computation. In consequence, a performance drop could only be observed for small grids. For sufficiently large grids, linear speed-ups were observed even on Astral 128 cores.

The next step was to assess the impact of communication overheads on the DFPS performance. Since Astral runs tens of user applications at a time, the consistency of computation time measurements had to be ensured by using nodes in the exclusive mode, however it was not possible to completely isolate communication operations. Timings of MPI operations were not as consistent, however certain trends could be identified. The results are presented in Figure 9.11.



(A) Observed data transfer

(B) Execution time breakdown (128 cores)

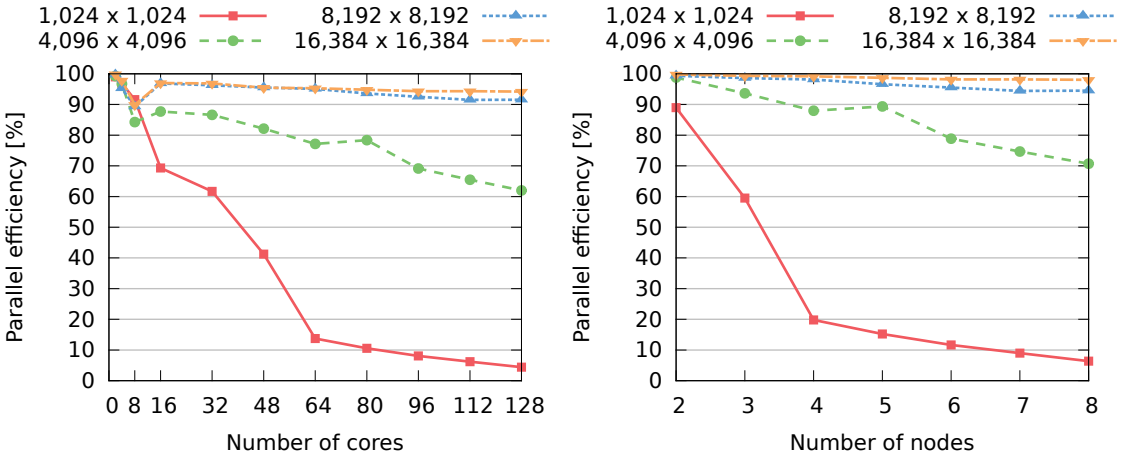
FIGURE 9.11: Impact of communication on DFPS performance on Astral (single precision). The noticeable drop in throughput is connected with the network topology. Communication overheads are significant for small grids, but become negligible for large ones.

For sufficiently large grids, communication was performed at transfer rates exceeding 30 Gbps (Figure 9.11A). Regardless of the grid size, a significant drop in the throughput was observed when using four or more Astral nodes. This was connected with the *fat tree* network topology, but could also be affected by network congestion caused by other applications running on Astral.

On smaller grids ($n \leq 4{,}096$), communication overheads account for a significant fraction of the total execution time even when using Infiniband interconnect (Figure 9.11B). However, the overheads decreased with the increasing grid size, and became negligible for sufficiently large grids even on 128 Astral cores.

Communication overheads had a direct reflection in the parallel efficiency observed for DFPS. On small grids, speed-ups decreased quickly with the increasing number

of cores due to the increasing cost of MPI transfers (Figure 9.12). A local minimum was observed when 8 cores were used (Figure 9.12 A) — it was connected with the previously discussed cache effects. Nevertheless, DFPS is highly scalable for sufficiently large grids: the speed-ups are close to linear even on 128 Astral cores.



(A) Per-core (B) Per-node (node = 16 cores)

FIGURE 9.12: Parallel efficiency of DFPS on Astral (single precision). Each node consists of two 8-core CPUs. When the Infiniband interconnect is used, DFPS is highly scalable for sufficiently large grids: the speed-ups are close to linear even on 128 cores.

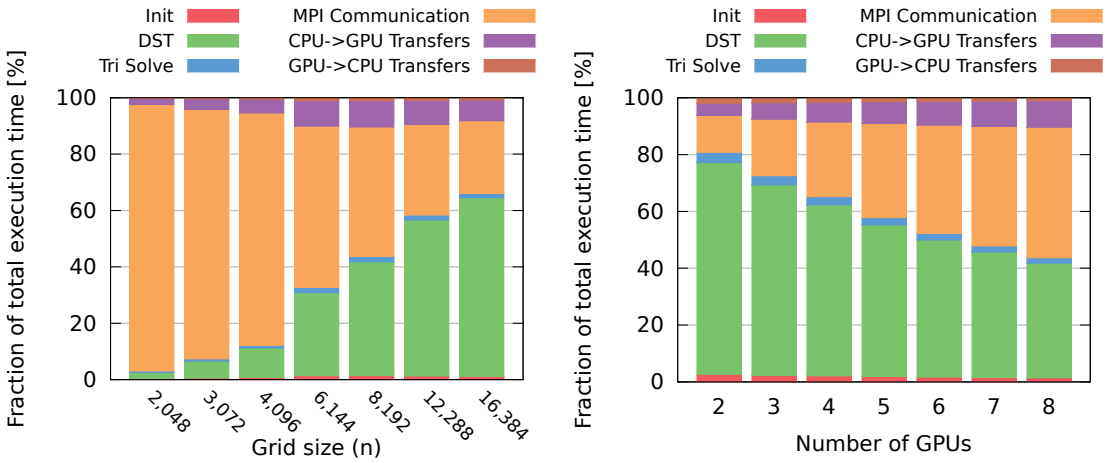
The performance observed on up to 16 cores was disturbed by two factors: constant CPU cache size for up to 8 cores, and the fact that MPI communication could be performed via shared memory, rather than the network. Therefore, it was sensible to treat each Astral node as a single processing element and to express the DFPS performance in terms of per-node parallel efficiency (Figure 9.12 B). The trends were similar to per-core results, but they were more consistent. The Distributed Fast Poisson Solver solved the 2D Poisson's Equation on a 16,384 × 16,384 grid in less than one minute using 128 Astral cores at 98% parallel efficiency.

9.4.5 Estimates for GPU Cluster with Infiniband

The final experiment should check the DFPS performance on the hybrid CPU-GPU parallel platform with Infiniband interconnect. This hardware configuration was not available, therefore only the expected performance is presented in this subsection.

The estimates are based on the performance model in Subsection 9.4.1, single GPU performance (Subsection 9.4.3), and Infiniband communication times observed in Subsection 9.4.4. The following assumptions were made: the GPU cluster consists of 8 Tesla C2050 GPUs (DST computation in single precision runs at 710 GFLOPS, tridiagonal solver runs at 3.6 GFLOPS), and PCI-Express transfers are executed at 6 GB/s (CPU→GPU) and 6.6 GB/s (GPU→CPU).

The estimated execution time breakdown is presented in Figure 9.13. The amount of data received from the GPU depends on the number of GPUs ($\frac{nm}{P}$ floating-point numbers), therefore GPU→CPU transfer time scales well with increasing P. However, the data sent to the GPU always consists of nm numbers, regardless of the number of GPUs — this has an adverse effect on scalability. In consequence, CPU→GPU transfers required significantly more time than copying data from the GPU to the CPU.



(A) 8 GPUs (B) Grid size: 8,192 × 8,192

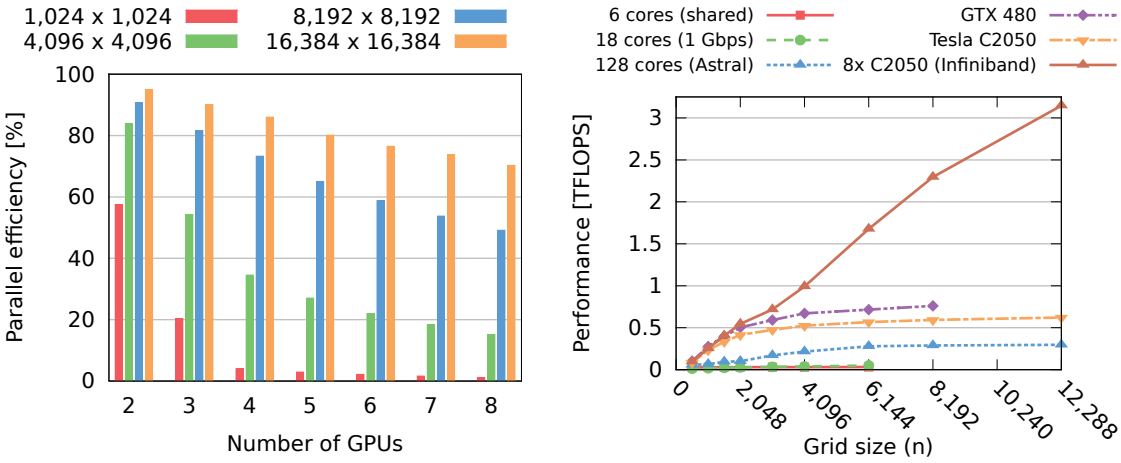
FIGURE 9.13: Estimated execution time breakdown of hybrid solver on GPU cluster consisting of 8 Tesla C2050 GPUs and Infiniband connectivity (single precision). Even when using Infiniband, the MPI communication can dominate the execution time. The amount of data transferred to each GPU is constant regardless of the number of GPUs. In consequence, CPU-GPU transfers may become more prominent when more GPUs are used, or larger problems are solved.

If 8 GPU cards were used, the MPI communication would clearly be the main performance limiting factor even for relatively large grids ($n \leq 8{,}192$). As the grid

size was increased, the time required for the Discrete Sine Transform would become more prominent, but even for $n = 16{,}384$ MPI communication would account for over 25% of the total execution time (Figure 9.13A).

If only two GPUs were used, communication overheads would be relatively small (Figure 9.13B). As expected, increasing the number of GPUs would result in a significant increase in the fraction of time spent on MPI communication. This is consistent with other experiments presented in this section, and would have an adverse effect on the hybrid DFPS parallel efficiency.

In Figure 9.14 the estimated performance of the Distributed Fast Poisson Solver on the GPU cluster with Infiniband interconnect is compared with other parallel platforms. The parallel performance of the hybrid DFPS would greatly depend on the grid size due to an increasing fraction of time spent on communication (Figure 9.14A). Furthermore, the increasing cost of MPI transfers would cause a steady decrease of parallel efficiency with the increasing number of GPUs. This would be less prominent in the case of sufficiently large grids — over 70% parallel efficiency is expected on 8 Tesla C2050 GPUs on a $16{,}384 \times 16{,}384$ grid.



(A) Hybrid solver efficiency

(B) Parallel platforms comparison

FIGURE 9.14: Estimated parallel performance of hybrid DFPS (A) and comparison with other parallel platforms (B) (single precision). Even with Infiniband, the parallel efficiency decreases due to communication overheads. A single commodity-grade GPU (GTX 480) is roughly 2.5 times faster than 128 CPU cores of Astral. A GPU cluster with Infiniband is likely to deliver an outstanding performance — over 3 TFLOPS with 8 Tesla C2050 cards.

The Distributed Fast Poisson Solver does not perform well on small-scale CPU clusters with Gigabit Ethernet interconnect: the performance up to only 50 GFLOPS was observed (Figure 9.14B). The large-scale CPU cluster with Infiniband interconnect was capable of delivering up to 300 GFLOPS on 128 cores. This could be increased to roughly 3 TFLOPS if all Astral cores were used. In comparison, the LINPACK benchmark runs at roughly 20 TFLOPS on Astral, therefore further improvement in performance is possible, e.g. by using a faster GEMM implementation from ATLAS or Intel MKL libraries.

In comparison to CPU-based parallel platforms, graphics cards offered a competitive performance, especially if the installation cost is considered. High instruction throughput on GPUs was achieved thanks to a very fast dense matrix-matrix multiplication in the CUBLAS library. On a single commodity-grade GPU (GTX 480), the solver run at up to 760 GFLOPS, i.e. 2.5 times faster than 128 CPU cores of Astral. The estimates indicate that on a GPU cluster with 8 Tesla C2050 cards and Infiniband interconnect a performance of more than 3 TFLOPS could be expected.

9.5 Conclusions

- In this chapter, the mathematical foundations for the FFT-based solver for 2D Poisson's Equation are presented. Unlike algorithms considered in the previous chapters, Fast Poisson Solvers are *direct* methods. FPS involves the Discrete Sine Transform and the solution of tridiagonal systems — multiple sources of parallelism in both steps were identified and discussed.

- This chapter focuses on Fast Poisson Solver implementations in distributed memory systems. The naive implementation involves two `Allgather` collective communication operations, but the initial analysis identified this as the main scalability limiting factor. An alternative implementation, requiring only one `Allgather`, was proposed and implemented. It benefits from better scalability (an optimal number of processing elements is increased from $\mathcal{O}(\log n)$ to $\mathcal{O}(\sqrt{n})$), but at the cost of a higher complexity of the DST step.

- The DST step has the highest time complexity and quickly becomes the dominant operation in the DFPS solver. All computation steps can be parallelised

and typically achieve close to linear speed-ups for sufficiently large problems. As the number of processing elements increases, the `Allgather` operation becomes more prominent and limits the scalability of DFPS.

- In systems based on Gigabit Ethernet interconnect, parallel efficiency decreases rapidly with the increasing number of processing elements due to communication overheads, but remains at an acceptable level for sufficiently large grids.

- GPU solvers reach the peak performance for large grids only: at least 2,048 × 2,048 elements in single precision and 1,024 × 1,024 in double precision, but the performance greatly exceeds that of the CPU solver: GTX 480 runs at up to 760 GFLOPS, which is over 20 times faster than a six-core Intel i7-980x.

- The additional computation time associated with switching to double precision arithmetic is significantly higher on the GPUs, especially on the cards designed for gaming, e.g. GTX 480. Nevertheless, the double precision GPU solver is almost 10 times faster on Tesla C2050 than the CPU implementation.

- Gigabit Ethernet is too slow for the hybrid DFPS implementation to run efficiently. In fact, when run on multiple GPUs, the total execution time becomes higher than when one GPU is used instead.

- The introduction of Infiniband interconnect greatly improves the DFPS scalability. To solve 2D Poisson's Equation on the grid consisting of 16,384 × 16,384 elements, the Distributed Fast Poisson Solver required less than one minute on 128 Astral cores at 98% parallel efficiency. This amounts to roughly 300 GFLOPS, which is still over 2.5 times slower than a single GTX 480 GPU.

- Replacing Gigabit Ethernet with Infiniband in the GPU cluster would lead to a significant improvement in parallel efficiency and overall performance. If all 1,280 Astral cores were used, then DFPS performance of up to 3 TFLOPS could be observed. The accurate performance model indicates that this performance could be exceeded with just 8 Tesla C2050 GPUs. This is a huge gain, considering significantly lower installation and running costs.

Chapter 10

Multi-GPU Solver for 2D Poisson's Equation

The previous chapter presented a highly scalable 2D Poisson solver based on Fourier analysis and designed for distributed memory systems. The algorithms considered in this chapter are adaptations of the same approach (cf. Section 9.2) designed for shared memory systems. Change in the memory architecture allows for more efficient communication and enables an order of magnitude faster Discrete Sine Transform computation. The use of shared memory limits the maximum number of processing elements (CPU cores, GPUs), therefore the main objective was to maximise the scalability, i.e. the ability to solve arbitrarily large problems in acceptable time.

This chapter is organised as follows. Section 10.1 introduces the FFT-based Fast Poisson Solver, discusses the related literature, and provides the complexity analysis. The details on several CPU- and GPU-based implementations are given in Section 10.2. This section also elaborates how the Discrete Sine Transform can be efficiently computed using the general Fast Fourier Transform operation. The results of numerical experiments are presented and discussed in Section 10.3. Finally, the conclusions are given in Section 10.4.

10.1 Fast Poisson Solver in Shared Memory Systems

This section describes differences and modifications to the implementation presented in the previous chapter. Subsection 10.1.1 presents the related work that motivated

some of the solutions employed in this chapter. Then, a discussion on the complexity of the Fast Poisson Solver (FPS) on different parallel platforms is provided in Subsection 10.1.2. Finally, Subsection 10.1.3 explains how the Fast Fourier Transform can be used to improve the complexity of the Discrete Sine Transform operation.

10.1.1 Related Work

There were several attempts to use the FFT in Poisson solvers, however in most cases, the 3D problem was addressed (Giraud et al., 1999; Serafini et al., 2005; Wu et al., 2014). These papers are discussed in more detail in the following chapter. A recent publication by Bleichrodt et al. (2012) presents results on a GPU-accelerated 2D Poisson solver used in a barotropic ocean model. To the best of our knowledge, all, even the most recently published GPU-based Poisson solvers are limited by the size of the device memory (Bleichrodt et al., 2012; Wu et al., 2014).

The performance of the Fast Fourier Transform is crucial for the performance of FFT-based Poisson solvers. The most commonly used approach is the algorithm proposed by Cooley and Tukey (1965), which follows the divide and conquer method. Van Loan (1992) provides a comprehensive description of this, and other algorithms for FFT. Due to a vast area of application, a number of high-performance FFT implementations have been created. Fastest Fourier Transform in the West (FFTW) is among the most prominent CPU libraries (Frigo and Johnson, 2005). The CUFFT library (NVIDIA, 2012b) is the GPU implementation based on the FFTW library and provides good performance, however it is not as mature and optimised as FFTW (Wu and JaJa, 2012; Wu et al., 2014).

Püschel and Moura (2007) presented optimised implementations of Discrete Sine and Cosine Transforms based on the Cooley-Tukey method. Since they require a significant coding effort and typically offer up to two-fold gain in performance, the solver presented in this chapter adopts the method of computing the Discrete Sine Transform with the FFT on the GPU. The same approach was used by Bleichrodt et al. (2012) who follow the method presented by Demmel (1996). The authors used the 2D DST-I and scaled the intermediate result with the corresponding eigenvalues. In contrast, in our work the 1D DST-I is used together with the tridiagonal solver.

In addition, we propose modifications to data packing and unpacking in DST-I, which allow us to further improve the performance.

10.1.2 Complexity Analysis

For an $n \times n$ grid, the problem size $N = n^2$, the sequential implementation of FPS requires $\mathcal{O}(N \log N)$ time and compares favourably with Preconditioned Conjugate Gradients, which requires $\mathcal{O}(N^{1.5})$ time. The sequential Multigrid solver requires $\mathcal{O}(N)$ time (which is optimal order-wise), however it is more difficult to parallelise. In the theoretical PRAM model (assuming *free* communication), the Multigrid solver runs in $\mathcal{O}(\log^2 N)$ steps on N processing elements, whereas the FPS solver can reduce this to $\mathcal{O}(\log N)$ steps, which is optimal (Demmel, 1996). However, communication is not free in real parallel systems, and indeed becomes a bottleneck in many applications.

The hybrid Fast Poisson Solver presented in the previous chapter was tailored for large distributed memory systems. To minimise the communication cost, thus maximise scalability, a trade-off had to be made: DST-I was computed using matrix-matrix multiplication resulting in $\mathcal{O}(\frac{N^{1.5}}{P})$ complexity on P processing elements.

In the presence of shared memory, collective message-passing communication is no longer required. In consequence, DST-I can be calculated in $\mathcal{O}(N \log N)$ time and the overall complexity on P processing elements is $\mathcal{O}(\frac{N \log N}{P})$. The trade-off is that P values are considerably lower than in the distributed memory implementation.

10.1.3 FFT-Based Fast Poisson Solver

The FFT-based Fast Poisson Solver, studied in this chapter, follows the pseudo-code in Algorithm 9.1, i.e. two Discrete Sine Transforms are computed (one forward, and one inverse), and a number of tridiagonal systems are solved. The latter step is calculated using the Thomas Algorithm, the same as in the previous chapter. This approach provides an optimal complexity on CPU-based platforms and allows for an efficient parallelisation on the GPU by solving each system in a separate thread.

The DST-I can be computed using the Fast Fourier Transform (Bleichrodt et al., 2012). The Discrete Fourier Transform (DFT) is defined as:

$$Y_j = \sum_{k=0}^{n-1} y_k e^{-\frac{2\pi jki}{n}}, \tag{10.1}$$

where i denotes the imaginary unit.

Extending the original input vector $\{x_k\}_{k=1}^{n}$ to have odd symmetry gives:

$$\mathrm{DFT}\left([0, x_1, x_2, \ldots, x_n, 0, -x_n, -x_{n-1}, \ldots, -x_1]\right) = \{F_k\}_{k=0}^{2n+1}. \tag{10.2}$$

Finally, the DST-I vector $\{X_k\}_{k=1}^{n}$ can be extracted with (Press et al., 2007, §12.4)

$$X_k = \frac{i}{2} F_k, \quad \text{where } 1 \leq k \leq n. \tag{10.3}$$

The inverse DST-I is calculated similarly, but requires scaling the result by $\frac{2}{n+1}$.

The DST-I step requires a significant amount of computation in comparison to the amount of data transferred — it is a *compute-bound* operation on most platforms. In contrast, the tridiagonal solver typically performs only 8 floating-point operations for each accessed value (4 or 8 bytes), resulting in low computation-to-communication ratio. In consequence, finding solutions to a tridiagonal system is bound by memory.

Depending on the computing platform, the performance of FPS may be bound by computation or memory transfers. Since the DST-I step requires $\mathcal{O}(N \log N)$ time, compared to $\mathcal{O}(N)$ in the case of the tridiagonal solver, DST-I typically dominates the execution time — it is expected that the solver will be compute-bound.

10.2 Implementation Details

This section provides details on several Fast Poisson Solver implementations proposed and compared in this chapter. These include the multi-threaded CPU solver (Subsection 10.2.1), the single GPU solver (Subsection 10.2.2), and the Streaming FPS Solver that can be extended to run on multiple GPUs (Subsection 10.2.3).

10.2.1 Multi-Threaded CPU Solver

The multi-core, shared memory systems can potentially benefit from coarse-grain parallelism (cf. Subsection 9.2.2). DST-I on separate vectors, and finding solutions to different tridiagonal systems can be calculated independently and performed in parallel using multiple threads. To implement such a multi-threaded CPU solver, the OpenMP 3.0 library (OpenMP Architecture Review Board, 2008) was used.

To perform the DST-I operation, the FFTW library (Frigo and Johnson, 2005) was used. FFTW supports real-to-real transforms, therefore DST-I can be computed directly. Furthermore, FFTW supports the fine-grain parallelism using OpenMP or POSIX Threads, however the initial experiments showed that a one thread per vector (coarse-grain) approach yields significantly better performance due to lower parallelisation overheads. The pseudo-code for the multi-threaded FPS is presented in Algorithm 10.1.

Algorithm 10.1 The multi-threaded CPU implementation of the 2D FPS Solver

```
1: Compute FFTW plan for a single vector DST-I transform    { Reused by all threads }
2: Calculate inverse DST-I on the source term    { One OpenMP thread per vector }
3: Solve tridiagonal systems    { Thomas Algorithm, one OpenMP thread per system }
4: Calculate the final solution    { Forward DST-I, one OpenMP thread per vector }
```

Another important design decision was the data layout: if DST-I is performed on grid columns, then tridiagonal systems are solved for each grid row (or vice versa) — depending on whether the grid is stored column-wise or row-wise, one of the operations benefits from a more cache-friendly access pattern. The initial experiments confirmed that the former layout (favouring DST-I) led to a better overall performance. This is consistent with the fact that DST-I has higher time complexity.

10.2.2 Single GPU Solver

Since the Fast Poisson Solver is expected to be compute-bound (cf. Subsection 10.1.3), it should be beneficial to run all the computation on the GPU. To calculate DST-I, the CUFFT library was used (NVIDIA, 2012b). Unlike FFTW, the CUFFT library does not currently support real-to-real transforms, therefore the method

presented in Subsection 10.1.3 had to be implemented explicitly. The tridiagonal systems were solved with the GPU implementation of the Thomas Algorithm (cf. Subsection 9.3.2). The modified implementation is presented in Algorithm 10.2.

Algorithm 10.2 The single GPU implementation of the 2D FPS Solver

1: Transfer the source term to the GPU
2: Compute CUFFT plan for the transform on a batch of vectors { Required once }
3: Calculate the inverse DST-I on the source term { In batches }
4: Solve tridiagonal systems { In batches}
5: Calculate the final solution { The forward DST-I, in batches }
6: Transfer the final solution back to the CPU

Computing DST-I with the general FFT requires data pre- and post-processing (both operations run in $\mathcal{O}(N)$ time). The way these operations are implemented has a significant impact on the overall performance. The approach followed in this study is based on the work of Bleichrodt et al. (2012), but introduces several modifications, which should allow for better performance, especially on larger grids.

Bleichrodt et al. (2012) used the 2D DST-I and scaling by eigenvalues, whereas the 1D DST-I and the tridiagonal solver was used in this study. This replaces two $\mathcal{O}(N \log N)$ time complexity passes (computing DST-I in the second dimension) with one tridiagonal solver pass of $\mathcal{O}(N)$ time complexity. Our approach is likely to yield better performance on sufficiently large grids.

As explained in Subsection 10.1.3, each vector of size n has to be expanded to $2n+2$ real numbers. Then the real-to-complex transform can be applied. The resulting vector contains $\lceil \frac{n}{2} \rceil + 1$ complex numbers with zero real part and consecutive DST-I values in the imaginary part.

If the right half of the values in Equation (10.2) were replaced with zeros, then the imaginary parts of the transform will be halved and the real parts will contain some arbitrary values (the latter can be safely ignored). Resetting auxiliary memory to zero with the `cudaMemset` routine and then copying contiguous memory areas corresponding to the input data leads to a significantly improved performance of the packing phase due to fully coalesced global memory access.

After the transform is calculated, imaginary parts have to be copied back to replace the contiguous input data (an in-place DST-I). Since the real and imaginary parts

are interleaved, the effective global memory bandwidth is halved due to the excess data loaded during coalesced memory transactions. The scaling factor of $\sqrt{\frac{2}{n+1}}$ is applied to make forward and inverse transforms identical, thus avoiding the cost of a separate scaling step. Figure 10.1 presents the DST-I transform in detail.

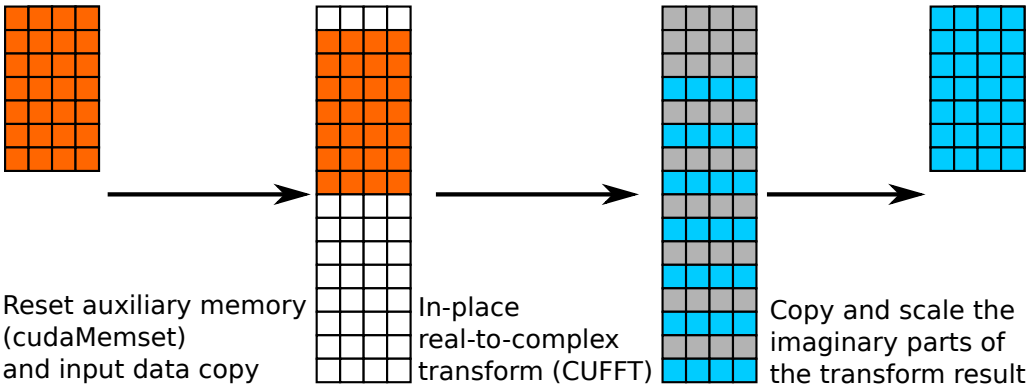


Figure 10.1: Computing DST-I on the GPU with CUFFT. Each cell corresponds to a single floating-point value. Colour legend: orange — input data, blue — output data, white — zeros, grey — arbitrary data (ignored).

The relatively small GPU global memory size is the main factor limiting the maximum size of a Poisson problem solved with FPS. In addition to memory used to store the input data, both the DST-I step and the tridiagonal solver require additional auxiliary space. Its size depends on the grid row or column length and the number of vectors or systems processed in parallel. To reduce the memory footprint, the solver can process data in smaller batches. The cost of multiple CUDA kernel dispatch is negligible, but the impact of a reduced workload (a batch may be too small to fully utilise the GPU) can be significant and is further explored in Subsection 10.3.3.

Moreover, the GPU solver imposes additional cost for communication between the CPU and the GPU. These data transfers are executed via PCI Express channels and to maximise the performance data on the CPU is stored in *page-locked* memory, allocated with `cudaHostAlloc` routine (NVIDIA, 2011, §3.2.4).

10.2.3 Streaming and Multi-GPU Solvers

Even with batched processing, the need to store the entire input data on the GPU limits the maximum Poisson problem size which can be solved with the GPU solver

presented in the previous subsection. The Streaming Fast Poisson Solver alleviates this limitation by dividing data into batches on the CPU rather than on the GPU. Then, one batch at a time is transferred to the GPU, the computation is performed, and then the data is transferred back, until all the batches are processed.

The downside of the streaming approach is that the data has to be transferred back and forth three times instead of one since the solver operations are performed on different data chunks (grid columns in DST-I, rows in the tridiagonal solver). This cost can be alleviated by using asynchronous memory transfers and the CUDA streams mechanism. The operations enqueued in a single stream are performed in the *first-in, first-out* order, however operations in different streams can be executed independently. The presence of asynchronous copy engines on the modern GPUs allows for overlapping data transfers with computation.

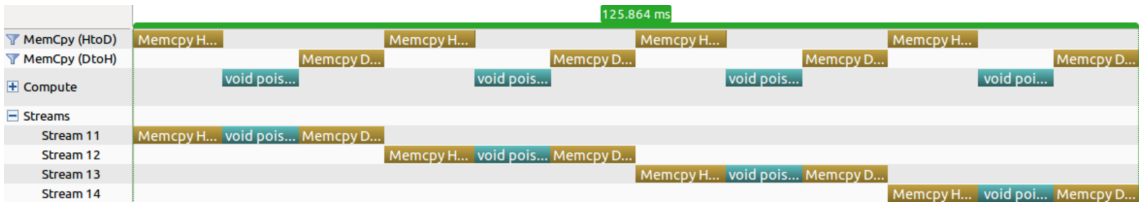
Unexpectedly, the attained level of overlap depends not only on the device capabilities (number of asynchronous copy engines), but also on the order in which the asynchronous requests are dispatched. The differences are explained in detail in the comprehensive on-line publication by Harris (2012).

The natural approach to CUDA call dispatch is presented in Algorithm 10.3. Upload, computation, and download requests are sent in the interleaved fashion, which leads to an unexpected performance degradation if a device is equipped with only one asynchronous copy engine. In this case, data can be transferred from, or to, the GPU, but not in both directions at the same time. In consequence, the upload of the second data chunk is scheduled *after* the download of the first chunk, but this can only be completed after the first chunk is processed, effectively inhibiting any computation-communication overlap (Figure 10.2 A).

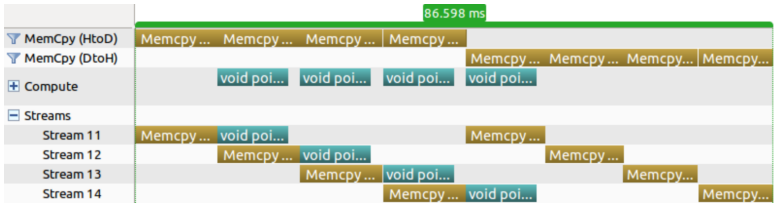
Algorithm 10.3 The natural request dispatch in the Streaming FPS Solver

```
1: for each batch do
2:     Request the upload of the data chunk to the GPU
3:     Request computation { DST-I or find solutions to tridiagonal systems }
4:     Request the download of the result from the GPU
5: end for
```

This issue can be addressed by grouping the requests by type, i.e. first all the uploads, then all the computation requests, and finally, all the downloads (Algorithm

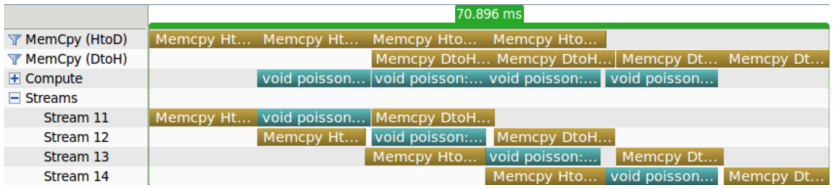


(A) Natural dispatch

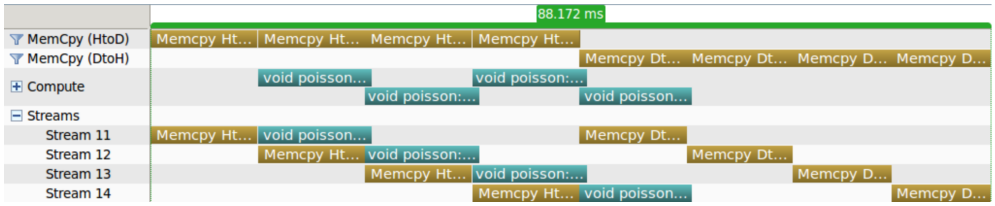


(B) Grouped dispatch

FIGURE 10.2: Profiler visualisation of kernel execution and data transfers on the GTX 480 GPU with one asynchronous copy engine. Issuing all CUDA calls in the natural order results in serial execution (A). On GPUs with one copy engine, computation can be overlapped with communication only when the CUDA calls are grouped by type (B). However, data can be transferred in one direction at a time only.



(A) Natural dispatch



(B) Grouped dispatch

FIGURE 10.3: Profiler visualisation of kernel execution and data transfers on the Tesla C2050 GPU with two asynchronous copy engines. The highest amount of overlap is observed when CUDA calls are dispatched in the natural order (A). Indeed, data transfers in both directions and computation run concurrently. If grouped dispatch is used on a GPU with two copy engines, data is transferred in one direction at a time only (B).

10.4). In this case, computation on a chunk of data can overlap with the upload of the next, or the download of the previous chunk. However, it is not possible to take advantage of duplex memory transfer via PCI Express (Figure 10.2 B). In consequence, CPU-GPU memory transfers become the performance limiting factor.

As expected, the natural dispatch leads to fully overlapped computation and data transfers on devices with two copy engines (Figure 10.3 A). In this case, all the operations are executed concurrently. In theory, the only overhead should be the upload of the first chunk, and the download of the last chunk. This suggests that using more CUDA streams may lead to a better performance, but at the cost of the higher memory requirement. This is further explored in Subsection 10.3.4.

Algorithm 10.4 The grouped request dispatch in the Streaming FPS Solver

```
1: for each batch do
2:     Request the upload of the data chunk to the GPU
3: end for
4: for each batch do
5:     Request computation { DST-I or find solutions to tridiagonal systems }
6: end for
7: for each batch do
8:     Request the download of the result from the GPU
9: end for
```

Unexpectedly, the grouped dispatch does not perform as well as the natural dispatch on devices with two copy engines. In this case, the download of the first chunk can only start after the upload of the last chunk, which effectively serialises upload and download operations (Figure 10.3 B). In consequence, no benefits from duplex PCI Express data transfers are observed. Interestingly, in this case, concurrent kernel execution was observed, although the performance improvement is negligible.

To ensure the high performance of our FPS solver on a broad range of GPUs, this peculiarity is taken into account — the best dispatch method is automatically selected based on device capabilities. The example execution for a large grid on a device with two copy engines is presented in Figure 10.4. Notably, the device is busy with computation most of the time and communication overheads are small.

Since the data is split into chunks on the CPU, the Streaming Fast Poisson Solver can be extended to use multiple GPUs for computation. In the case of homogeneous

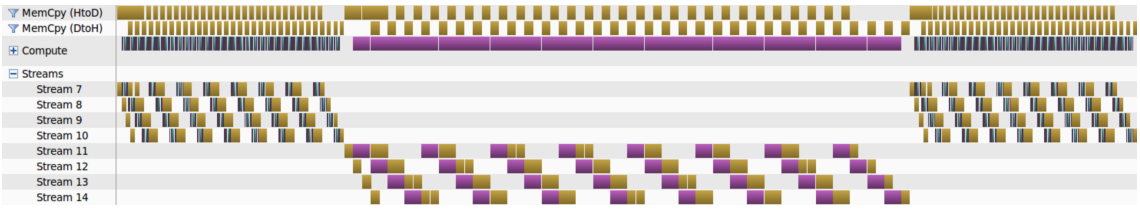


Figure 10.4: The streaming processing of the large grid ($n = 32{,}767$) on Tesla C2050. Despite the fact that the problem is too large to fit into global memory and data has to be processed in chunks, the device is engaged in computation almost all the time (the 'Compute' row). On GPUs with two copy engines, almost all CPU-GPU data transfers (yellow blocks) can execute concurrently (overlap) with computation.

multi-GPU platforms, a simple Round-Robin load balancing is sufficient. In each step, each GPU receives roughly the same amount of data that can be processed independently from other devices. This approach does not impose any additional overhead and typically should result in a close to linear speed-up with respect to the number of GPUs. This is further explored in Subsection 10.3.5.

In the presence of heterogeneous GPU resources, the proposed streaming solver can be enhanced with an automatic load balancing based on device theoretical capabilities or mini-benchmarks. This extension is not covered in the experiments section and is considered as a further research direction.

10.3 Numerical Experiments

In this section, several FFT-based FPS implementations were compared using performance models described in Subsection 10.3.1: the multi-threaded FPS (Subsection 10.3.2), FPS on a single GPU (Subsection 10.3.3), the Streaming FPS (Subsection 10.3.4), and the multi-GPU FPS (Subsection 10.3.5). Finally, Subsection 10.3.6 presents the comparison with the implementation by Bleichrodt et al. (2012).

Technical details of two CPU models (Intel i7-980x and Intel Xeon E5462) and three GPU models (Tesla C1060, Tesla C2050, GTX 480) used in the experiments are summarised in Appendix B in Tables B.1 and B.2, respectively. All machines used in the experiments were equipped with 12 GB of main memory, were running the Ubuntu 12.04 Server (64-bit) operating system, and were using CUDA driver

304.88. The CPU code was compiled with `gcc` version 4.6.3, and the GPU code was compiled with `nvcc` release 5.0, V0.2.1221 and CUDA Toolkit 5.0.35. Unless specified otherwise, all experiments were run in single and double floating-point arithmetic, were repeated ten times, results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5% (less than 1% in most cases).

10.3.1 Performance Models

To ensure reliable performance measurement and fair comparison between the algorithms the following performance models were used (based on Wu et al., 2014).

$$F_{dst}(n,m) = \frac{m \cdot 5n \log_2 n}{t_{dst} \cdot 10^9} \qquad [GFLOPS] \qquad (10.4)$$

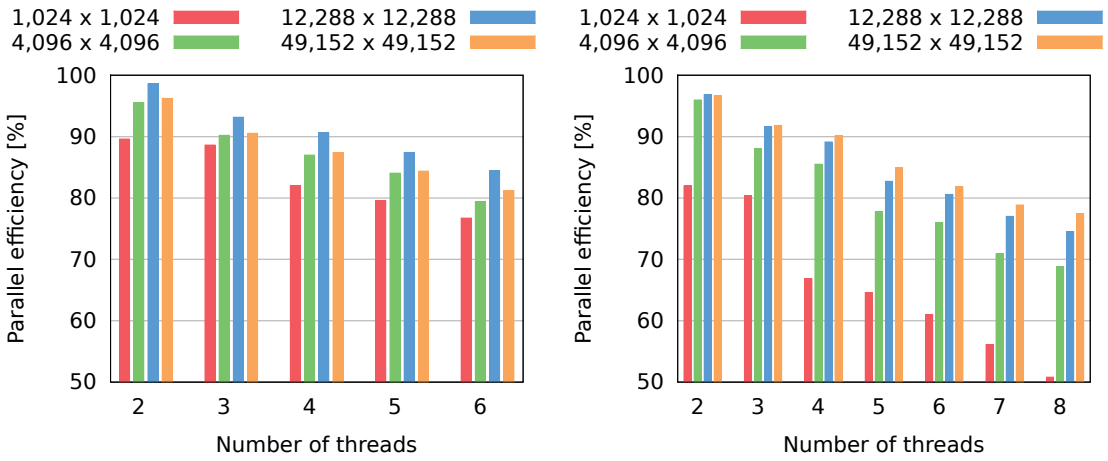
$$F_{tri}(n,m) = \frac{n \cdot 8m}{t_{tri} \cdot 10^9} \qquad [GFLOPS] \qquad (10.5)$$

$$F_{overall}(n,m) = \frac{nm \cdot (10 \log_2 n + 8)}{t_{overall} \cdot 10^9} \qquad [GFLOPS] \qquad (10.6)$$

Here, n and m denote the number of rows and columns in the grid, respectively. In all the experiments, only square grids were considered, i.e. $n = m$.

10.3.2 Multi-Threaded CPU Solver

In general, the Discrete Sine Transform was susceptible to parallelisation and yielded high parallel efficiency on both CPUs (Figure 10.5). The performance increased with the grid size due to less prominent parallelisation overheads: OpenMP dispatch costs and higher rate of CPU cache misses. The former cost is negligible for large grids, however the latter explains the steady decrease in performance on both CPUs with increasing number of threads, regardless of the problem size. The inefficient use of CPU cache also explains why there is a drop in performance for a 49,152 × 49,152 grid on Intel i7-980x (Figure 10.5 A), but not on Intel Xeon E5462 (Figure 10.5 B). Only the former CPU is equipped with per-core L2 cache (256 KB) which is fully utilised for smaller problems, but each DST-I data chunk is 192 KB for the largest instance, and thus 25% of L2 cache space is wasted. Despite these limitations, parallel efficiency remained at a high, 75–85%, level for sufficiently large grids.



(A) Intel i7-980x (six cores)

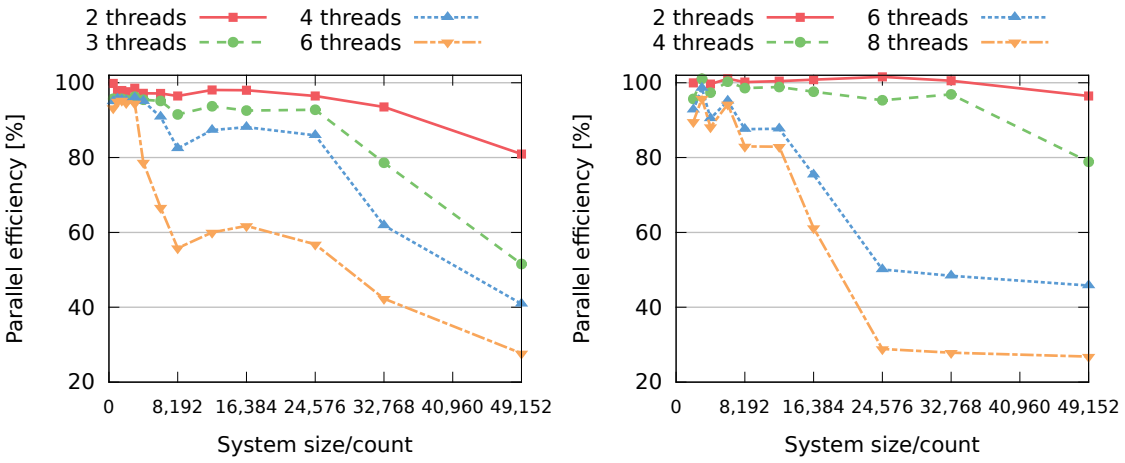
(B) Intel Xeon E5462 (2× four cores)

FIGURE 10.5: Parallel efficiency of the multi-threaded DST-I (single precision). The efficiency decreases with the increasing thread count due to parallelisation overheads. Ineffective use of the L2 cache on Intel i7-980x causes lower efficiency on the largest grid.

The performance of the multi-threaded tridiagonal solver was less consistent and varied greatly depending on the CPU and the thread count (Figure 10.6). The main cause was the data layout: when DST-I is performed on grid columns, then tridiagonal systems are solved for each grid row. Since the data is stored in the column-major order, DST-I benefits from the high cache hit rate, at the cost of the degraded tridiagonal solver performance.

The column-major grid layout caused the tridiagonal system coefficients to be strided in memory, i.e. the distance between consecutive values was proportional to the system size. This had an adverse effect on performance, since L3 cache miss ratio increased quickly with the grid size. This explains the parallel efficiency drop, regardless of the CPU model and the problem size (Figure 10.6).

Similar to the multi-threaded DST-I case, the presence of L2 cache on Intel i7-980x had an unexpected impact on performance (Figure 10.6 A). When a particular piece of data was requested, typically a larger chunk would be fetched to the L2 cache. For smaller problems that chunk would contain a number of values which would be requested next and the consecutive reads would be faster. As the problem size increased to 8,192 elements (32 KB in single precision — the L1 cache size), the L2 cache hit ratio decreased. Once the problem size exceeds the L1 cache size, some



(A) Intel i7-980x (six cores)

(B) Intel Xeon E5462 (2× four cores)

FIGURE 10.6: Parallel efficiency of the multi-threaded tridiagonal solver (single precision). A significant decrease in parallel performance with increasing problem size and the number of threads is caused by non-optimal data layout leading to a high cache miss ratio.

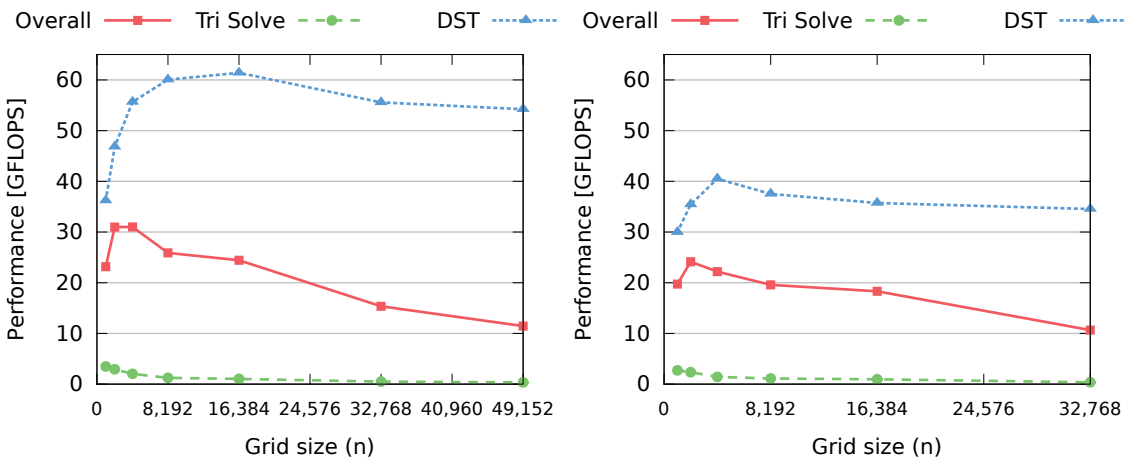
data pre-fetched to L2 cache became useful again. In consequence, there was an observable local minimum in parallel efficiency for systems with 8,192 elements.

When more than four cores of Xeon E5462 were used, a significant performance degradation was observed for large grids (Figure 10.6 B) due to the fact that there is no cache spanning two CPUs. Instead, each four-core CPU has two 6 MB L3 caches leading to a higher cache miss ratio when more than four threads were in use.

Despite differences in parallel efficiency of each step, the multi-threaded FPS Solver showed similar performance characteristics on both CPUs: the peak was observed when the speed of DST-I was the highest, i.e. for medium and large grids (Figure 10.7). For large grids the performance of DST-I remained constant, but decreasing tridiagonal solver speed caused a steady degradation of the overall performance. The solver achieved the peak performance of more than 30 GFLOPS in single precision (Figure 10.7 A), and more than 25 GFLOPS in double precision (Figure 10.7 B).

10.3.3 Single GPU Solver

The experiments confirmed the expected high performance of the DST-I adaptation using CUFFT (Figure 10.8). The batch size had a limited impact on the performance



(A) Single precision (B) Double precision

FIGURE 10.7: The performance of the multi-threaded 2D Fast Poisson Solver on all six cores of Intel i7-980x. The solver runs at up to 30 GFLOPS (out of roughly 60 GFLOPS maximum measured performance). The peak overall performance is observed for relatively small grids, then it decreases due to the low tridiagonal solver performance.

— indeed, the speed of large DST-I ($n = 8{,}192$) was constant on all GPU cards (Figure 10.8 B). The high speed was also observed for small DST-I ($n = 512$), except for small batch sizes ($< 1{,}024$) due to device under-utilisation (Figure 10.8 A).

The Discrete Sine Transform is compute-bound, therefore computation speed should be correlated with the GPU peak performance. Indeed, on both devices with Compute Capability 2.0 20% of the theoretical peak performance was observed (up to 240 GFLOPS on GTX 480). This fraction is smaller on Tesla C1060 (Compute Capability 1.3) and amounts to roughly 12% of the theoretical peak performance — an older Tesla C1060 lacks the upgrades to memory performance (implicit global memory caching, larger register count and shared memory size) introduced in Compute Capability 2.0. In consequence, DST-I became memory-bound on Tesla C1060. The difference in Compute Capabilities is also the reason why GTX 480 and Tesla C2050 obtain 25% higher performance for larger grids, while Tesla C1060 yields similar performance, regardless of the DST-I size.

Unlike the Discrete Sine Transform case, the batch size had a critical impact on the tridiagonal solver performance (Figure 10.9). This was directly connected with the device utilisation since only one thread was run per tridiagonal system. On devices

with Compute Capability 2.0 the number of thread blocks should be at least double the number of streaming multi-processors (NVIDIA, 2011) — over 7,000 threads when using 256 threads per block. This is consistent with the fact that GTX 480 and Tesla C2050 achieved the peak performance when 8,192 systems were solved.

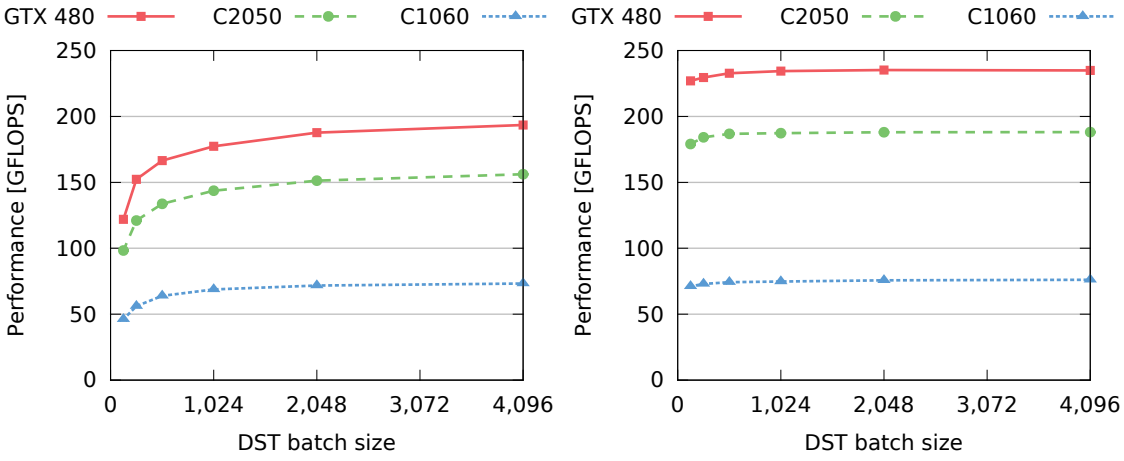
Tesla C1060 (Compute Capability 1.3) also obtained the peak performance when $n \geq 8{,}192$, but since it has half the cores running at higher frequency than the other two cards, better occupancy is attained for smaller grids. In consequence, for smaller systems ($n \leq 4{,}096$) Tesla C1060 delivered the same performance as Tesla C2050 even though it has significantly lower specifications.

Unlike DST-I, the tridiagonal solver is memory-bound and the performance was proportional to the theoretical peak memory bandwidth. This explains why the observed performance was identical for small (Figure 10.9 A) and large (Figure 10.9 B) tridiagonal systems. When at least 8,192 tridiagonal systems were solved, then our implementation obtained 45–55% of the theoretical peak memory bandwidth, regardless of system size or device Compute Capability.

The next step was to assess the overall performance of the FPS solver on the GPU. Based on the results from previous experiments, the DST-I batch size was set to 512 columns and the tridiagonal solver batch was set to 8,192 rows. Since GTX 480 had only 1.5 GB of memory, the largest problems could not be solved. Moreover, the tridiagonal solver batch size had to be reduced for the largest grids on Tesla C2050. The overall performance was measured according to Equation (10.6) taking the CPU-GPU transfer time into account.

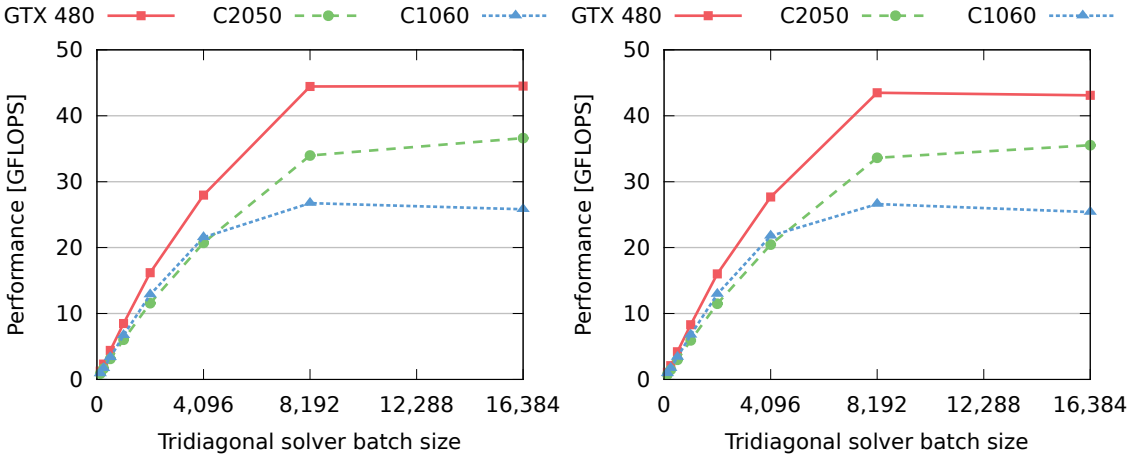
The GPU implementation ran much faster than the multi-threaded CPU version (Figure 10.10). When single precision arithmetic was used, the GPU solver performance increased with the problem size to achieve the peak when $n \geq 4{,}096$ (Figure 10.10 A). The decrease in performance observed for the largest grids was connected with the reduced tridiagonal solver batch size due to the limited global memory size. Overall, the GPU version running on GTX 480 attained the instruction throughput of 205 GFLOPS, compared to roughly 30 GFLOPS on a high-end, six-core CPU. Depending on the problem size, the speed-ups varied between 3.5 and 8 times.

The peak performance on the GPU was achieved for even smaller grids ($n \geq 2{,}048$), when double precision was used (Figure 10.10 B). In this case, much smaller differences between the CPU and the GPU could be observed, due to significantly lower



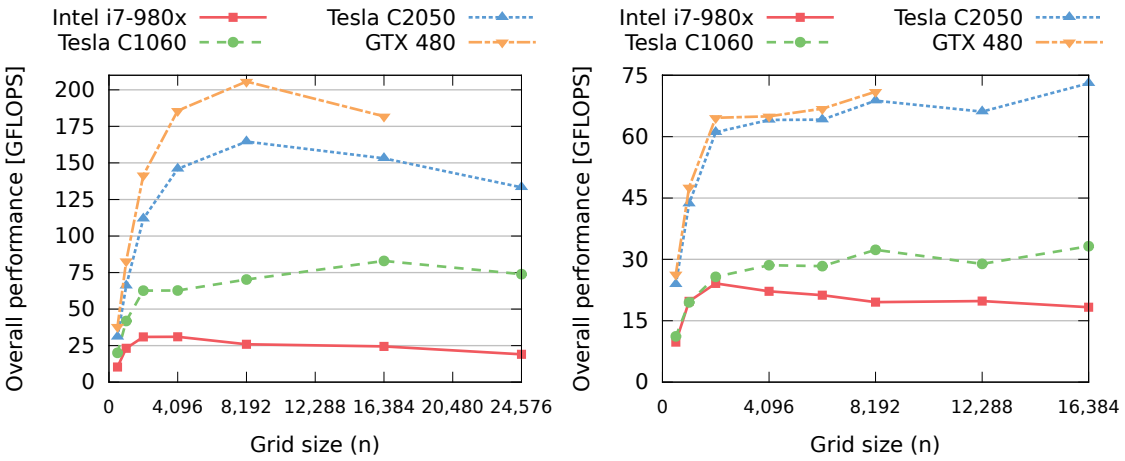
(A) Small DST-I ($n = 512$) (B) Large DST-I ($n = 8{,}192$)

FIGURE 10.8: Impact of the batch size on the DST-I operation performance. The speed is proportional to the theoretical peak performance of each device (the operation is compute-bound).



(A) Small systems ($n = 512$) (B) Large systems ($n = 8{,}192$)

FIGURE 10.9: Impact of the batch size on the tridiagonal solver performance. The speed increases significantly until the GPUs are fully utilised (batch of 8,192). The performance is proportional to the maximum global memory bandwidth of each device (operation is memory-bound). The size of tridiagonal systems has a negligible impact on performance.



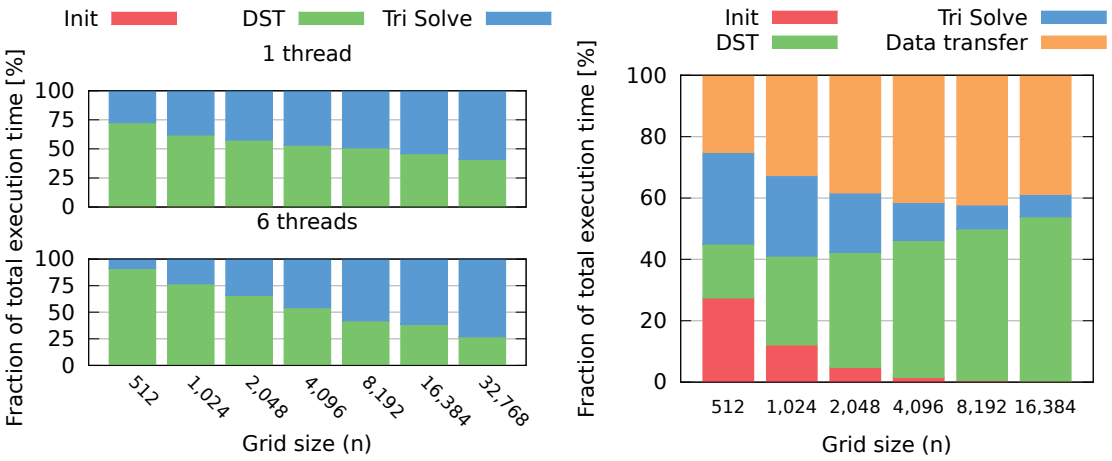
(A) Single precision (B) Double precision

FIGURE 10.10: The performance of 2D Fast Poisson Solver on a single GPU, which is proportional to the theoretical peak performance of each device; lower in case of Tesla C1060 due to lower Compute Capability. All three GPUs are significantly faster than the CPU implementation, especially in single precision.

double precision performance of the GPUs — roughly two times slower on Tesla C2050, significantly more on gaming cards (e.g. GTX 480) and older devices (Tesla C1060). This explains why GTX 480 and Tesla C2050 yielded similar overall performance, even though the former is roughly 20% faster in single precision. Overall, the fastest GPU versions were capable of delivering more than 73 GFLOPS, compared to roughly 24 GFLOPS on Intel i7-980x. Depending on the problem size, the speed-ups varied between 2.3 and 4 times.

To identify performance limiting factors on both CPU and GPU, the time required for each algorithm step was measured (Figure 10.11). The fraction of time spent on solving the tridiagonal systems increased significantly with the problem size (Figure 10.11 A) due to the increasing cache miss ratio. This became even more prominent when multiple threads were used — the tridiagonal solver could take up to 75% of the total execution time. In consequence, this step became a performance bottleneck on the CPU. However, rearranging data to improve the cache hit ratio would cause similar symptoms in the DST-I step, effectively leading to even worse performance.

The fraction of the execution time spent on solving tridiagonal systems decreased in GPU implementations (Figure 10.11 B). Indeed, it was DST-I and CPU-GPU data



(A) CPU (Intel i7-980x) (B) GPU (Tesla C2050)

Figure 10.11: Execution time proportions in the 2D FPS Solver (single precision). On the CPU, the tridiagonal solver becomes the dominant operation. In contrast, DST-I and CPU-GPU data transfers are the most prominent operations on the GPU.

transfers that became the most demanding steps. For the largest grids ($n \geq 16{,}384$), the former operation tended to dominate. This is consistent with the $\mathcal{O}(N \log N)$ time complexity of DST-I, compared to $\mathcal{O}(N)$ time required for data transfers.

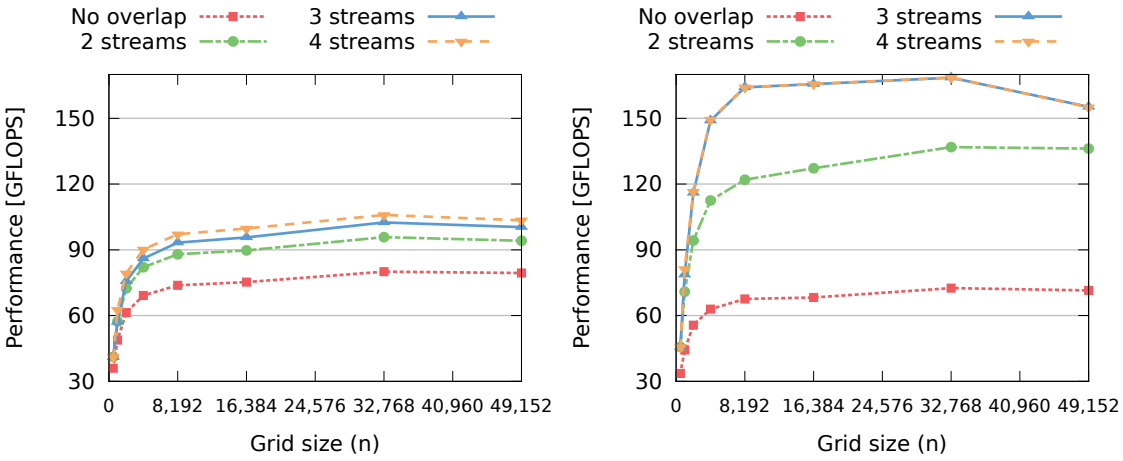
A further improvement in performance is possible, if DST-I is computed with an optimised hand-crafted algorithm based on the Püschel and Moura (2007) approach, rather than the general FFT. The CPU-GPU data transfers were executed at close to the optimal speed (6 and 6.6 GB/s for upload and download, respectively, on cards connected with PCI Express 2.0) and could be further improved if more recent PCI Express version was used (two times faster with PCI Express 3.0).

10.3.4 Streaming GPU Solver

The greatest shortcoming of the solution presented in the previous subsection, and indeed most GPU implementations of the Poisson solver presented in the literature, is that the maximum problem size is limited by the GPU global memory size, which is typically significantly smaller than the main memory size. The streaming GPU solver (cf. Subsection 10.2.3) allows us to solve arbitrarily large Poisson problems, however at a certain additional communication cost.

Due to concurrent kernel execution and asynchronous CPU-GPU data transfer supported by the modern GPUs, it is possible to partially hide the communication cost. The first step was to estimate the amount of overlap attainable on different GPUs.

All the experiments described in this subsection, were conducted on all three devices listed in Table B.2 in Appendix B. Results for an older generation GPU (Tesla C1060) showed that no overlapping was achieved, regardless of the number of CUDA streams used. Even though all kernel calls and memory transfers were performed in asynchronous manner with respect to the CPU, the execution on the GPU is completely serialised, because devices with Compute Capability 1.3 do not support overlapping for transfers of non-linear memory, e.g. `cudaMemcpy2DAsync` (NVIDIA, 2011, §3.2.5.2). In consequence, the results for Tesla C1060 are omitted.



(A) GTX 480 (1 copy engine)

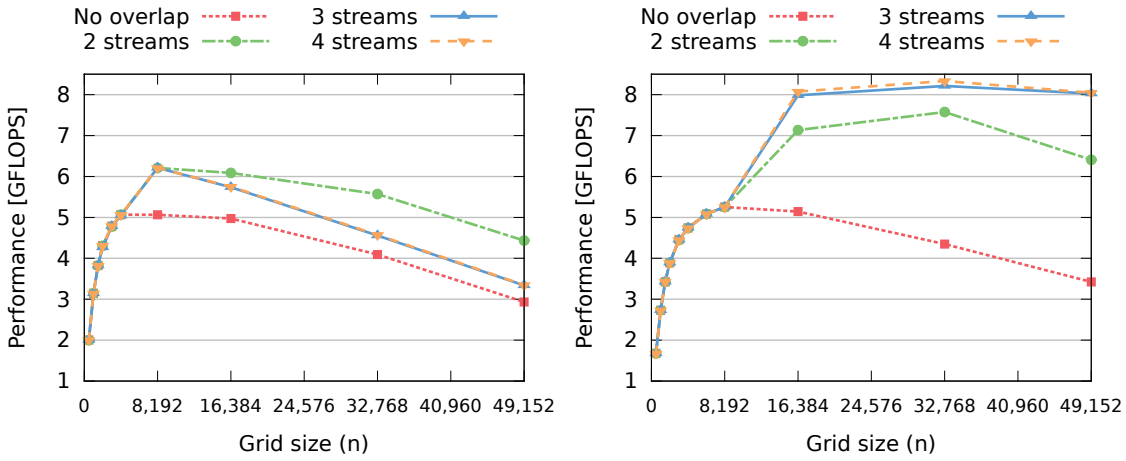
(B) Tesla C2050 (2 copy engines)

Figure 10.12: Performance of DST-I run on multiple CUDA streams (single precision). The results include CPU-GPU memory transfers. Using two, or three CUDA streams results in more overlapping and better performance, especially on devices with two copy engines.

Increasing the number of CUDA streams resulted in better DST-I speed, however the differences in performance varied greatly depending on the number of asynchronous copy engines (Figure 10.12). Significant increase of the speed can be observed even when only two streams were used. Introducing the third stream resulted in further significant improvement on Tesla C2050 (Figure 10.12B), however there was only a small increase on GTX 480 (Figure 10.12A). This is connected with the fact that the amount of overlap on devices with one copy engine is limited, as discussed in

Subsection 10.2.3. Adding more CUDA streams resulted in limited (GTX 480), or no further increase (Tesla C2050) in the DST-I performance. Using more CUDA streams is connected with increased memory requirements, therefore using three streams is the optimal choice for this application.

The performance characteristics of the streaming tridiagonal solver are different than that in the DST-I case (Figure 10.13). For the best performance, the tridiagonal solver batch size has to be set to at least 8,192 rows. In consequence, no overlap could be achieved for $n \leq$ 8,192, since only one batch was to be processed and other streams were idle. The diminishing benefits from the global memory caching with the increasing tridiagonal system size led to a steady decrease in tridiagonal solver performance, similar to that observed on the CPU. However, for large grids ($n >$ 8,192) the benefits from overlapping on Tesla C2050 were as expected (Figure 10.13B): a significant increase in the performance when the second and the third CUDA streams were added, negligible benefit from the fourth stream. As in the DST-I case, the optimal choice is to use three CUDA streams in the streaming tridiagonal solver.



(A) GTX 480 (1 copy engine)

(B) Tesla C2050 (2 copy engines)

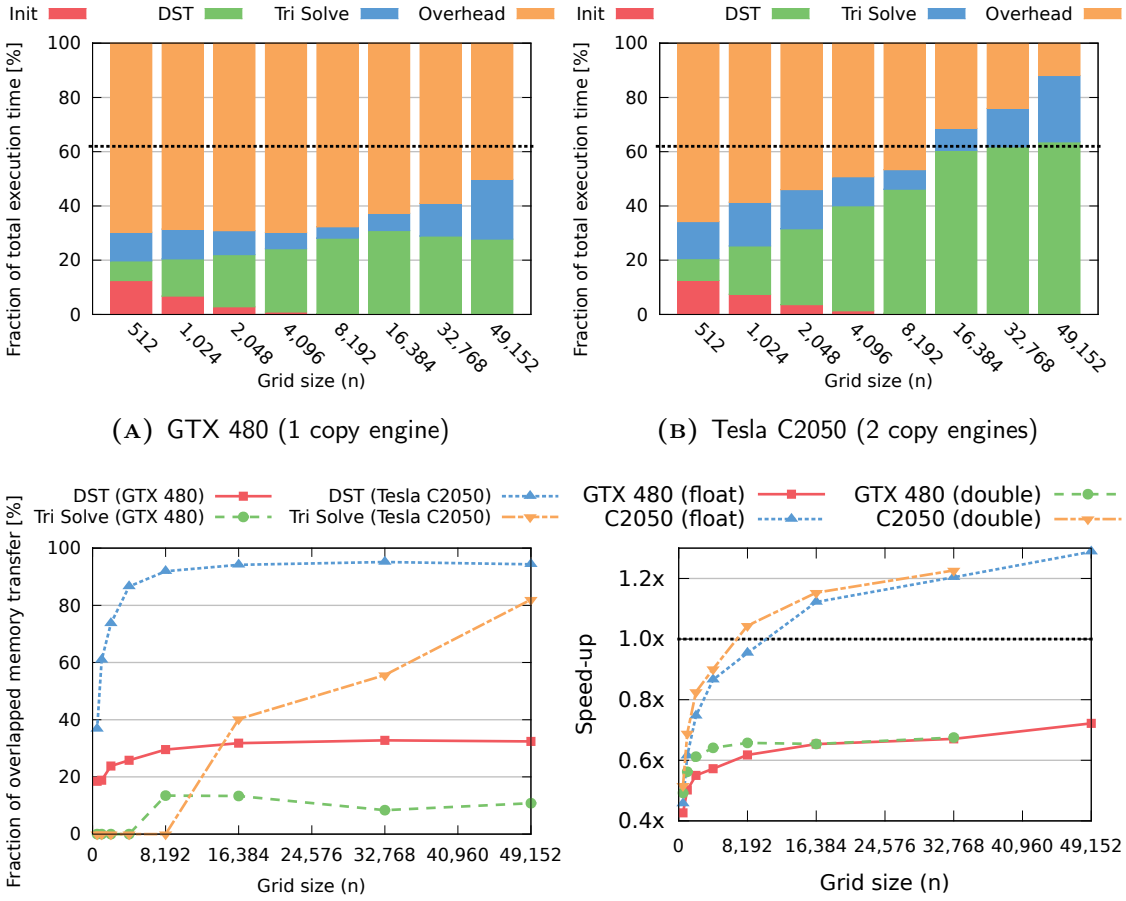
FIGURE 10.13: Performance of the tridiagonal solver executed on multiple CUDA streams (single precision). The results include CPU-GPU memory transfers. Using two or three CUDA streams results in more overlapping and better performance, especially on devices with two copy engines. However, using more streams increases the memory usage — the batch size had to be reduced leading to the lower performance on GTX 480.

Due to small global memory in GTX 480 (1.5 GB), the results are affected by the need to reduce the tridiagonal solver batch size to accommodate larger grids and more streams. In consequence, using three or four CUDA streams led to worse performance than when two streams were used instead (Figure 10.13 A). The decreasing batch size was also the reason for a more noticeable performance drop for larger grids. Finally, the initial batch size was set to 4,096, therefore the benefit from overlapping could be observed for $n = 8{,}192$. Nevertheless, due to the proposed streaming mechanism, it is possible to solve Poisson problems with $49{,}152 \times 49{,}152$ points (9 GB of input data) using a graphics card with only 1.5 GB memory.

The next step was to assess the impact of additional overheads on the whole streaming solver and compare it with the non-streaming version considered in the previous subsection. The overheads connected with streaming processing could amount to up to roughly 65% of the total execution time and were most significant on small grids (Figure 10.14). The fraction of overlapped memory transfers was increasing with the grid size (Figure 10.14 C). The maximum amount of attainable overlap was determined by the number of asynchronous copy engines: on GTX 480 (one copy engine) only up to 33% or 17% of memory transfers can be overlapped with DST-I or tridiagonal solver computation, respectively. When two copy engines were present (Tesla C2050), these fractions increased to 95% and 82%, respectively.

The fraction of overlapped memory transfers had a direct reflection in the execution time proportions: on GTX 480 the fraction of communication overhead slowly decreased to reach 50% for largest grids (Figure 10.14 A). In consequence, the device did not perform any calculations for roughly half of the total execution time. The presence of the second asynchronous copy engine (Tesla C2050) alleviated this limitation. Indeed, the proportion of communication overhead quickly decreased to reach only 12% for the largest grids (Figure 10.14 B). This is significantly better than 35–40% of time spent on CPU-GPU communication in the non-streaming solver (Figure 10.11 B). Whenever the 'Overhead' bar ends above the black dotted line, the streaming solver actually uses less time for communication than the non-streaming implementation, even though it transfers three times more data.

The fraction of communication overhead had a direct impact on how the streaming solver performance compared with the non-streaming counterpart. Indeed, the streaming solver on GTX 480 spent more time on non-overlapped communication,



(A) GTX 480 (1 copy engine)

(B) Tesla C2050 (2 copy engines)

(C) Attained level of overlapping

(D) Speed-up over the non-streaming solver

FIGURE 10.14: Execution time proportions in the 2D Streaming Fast Poisson Solver and the impact of streaming processing on overall performance (three CUDA streams, single precision). The dotted line approximates the communication overhead in the non-streaming FPS. The streaming FPS on Tesla C2050 is faster than the non-streaming counterpart even for relatively small grids due to the high level of overlapping. This is not attainable on GTX 480 with only one copy engine.

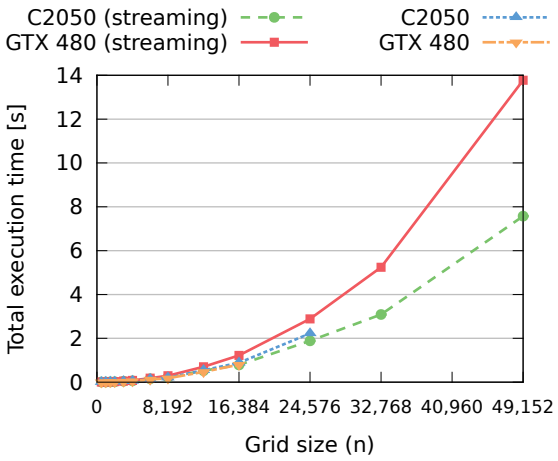
regardless of the grid size and the non-streaming implementation is over 30% faster (Figure 10.14 D). In contrast, on devices with two asynchronous copy engines (Tesla C2050), the fraction of non-overlapped transfers became small for $n \geq 8{,}192$, allowing the streaming implementation to run almost 30% faster than the non-streaming version. These results hold for computation in both single and double precision.

In the final experiment, the Streaming FPS solver was compared with the CPU and non-streaming GPU implementations (Figure 10.15). There were no significant differences in the total execution time for small grids ($n \leq 8{,}192$), regardless of device, precision, or whether streaming was used or not.

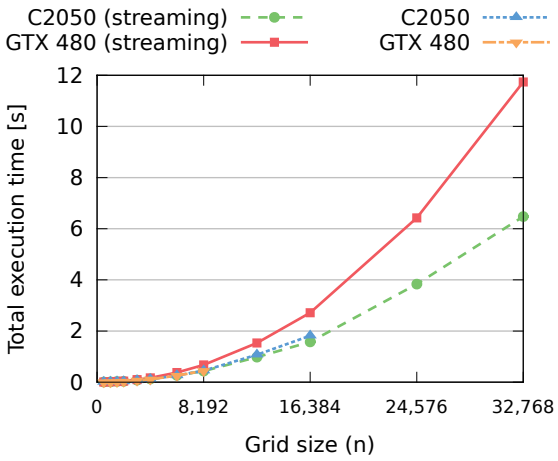
Both implementations were capable of solving problems with $n \leq 16{,}384$ in less than a second in single precision (Figure 10.15 A). As the grids got larger, the differences between streaming and non-streaming implementations and between devices with one or two asynchronous copy engines became more prominent. The Streaming FPS was noticeably slower on GTX 480, and slightly faster on Tesla C2050, due to a higher level of overlap attained on the latter device. The Streaming FPS solver on Tesla C2050 is significantly faster than on GTX 480 for the same reason, even though the latter device has better peak instruction throughput. For largest grids the streaming version ran roughly two times faster on Tesla C2050 than on GTX 480, due to the second asynchronous copy engine that allows us to overlap almost 90% of the CPU-GPU memory transfers. The fastest implementation (the Streaming FPS on Tesla C2050) could solve Poisson problems with one billion variables in roughly 3 seconds, or with 2.4 billion variables in less than 8 seconds (single precision).

The same performance characteristics could be observed when double precision was used, however the total execution times were roughly doubled (Figure 10.15 B). The fastest implementation (the Streaming FPS solver on Tesla C2050) could solve Poisson problems with one billion variables (8 GB of input data) in just over 6 seconds.

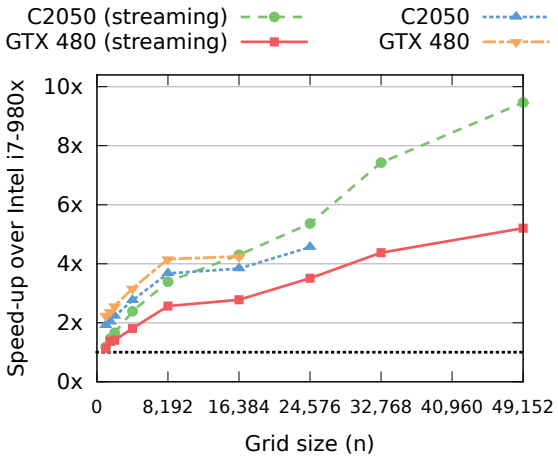
Regardless of the grid size and the precision used, GPU solvers were consistently faster than a highly optimised implementation run on a six-core Intel i7-980x CPU. Non-streaming solvers achieved speed-ups up to a factor of 4.5 in single precision (Figure 10.15 C), and up to a factor of 2.5 in double precision (Figure 10.15 D). The streaming solver on GTX 480 was the slowest due to limited memory transfer overlapping, however it still achieved speed-ups of a factor of over five (single precision),



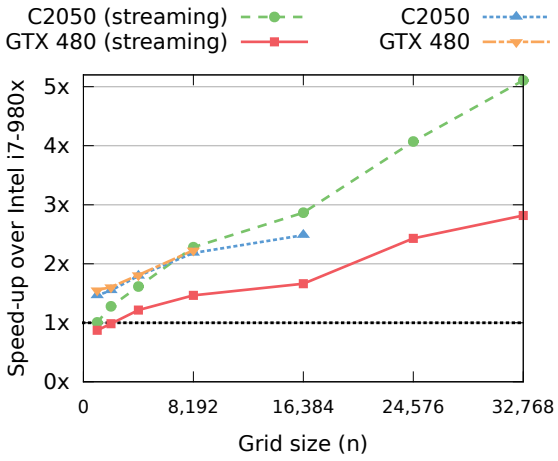
(A) Execution time (single precision)



(B) Execution time (double precision)



(C) Speed-up over CPU (single precision)



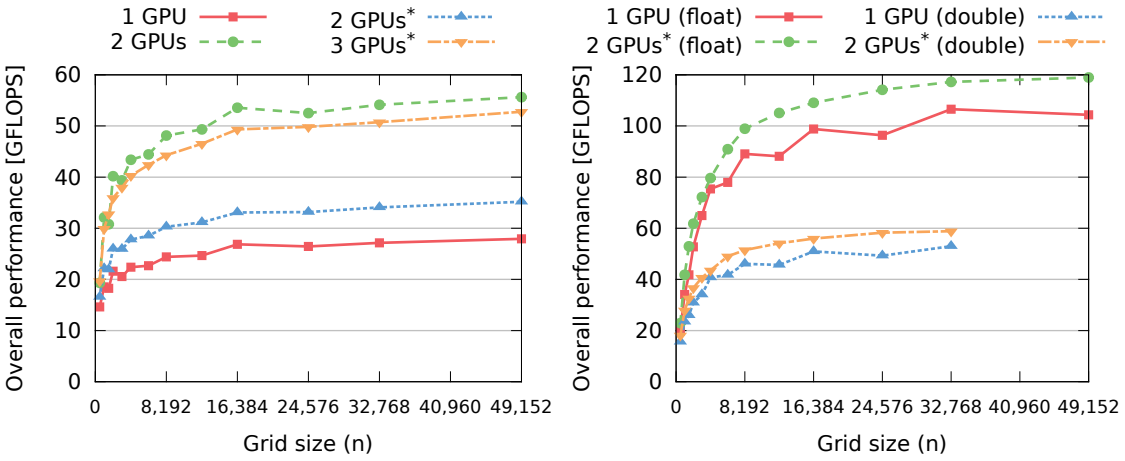
(D) Speed-up over CPU (double precision)

Figure 10.15: Performance of 2D Fast Poisson Solver implementations. Streaming GPU version used three CUDA streams. CPU-GPU data transfer time was taken into account when calculating execution times and speed-ups over the CPU implementation.

or three (double precision) for the largest grids. Regardless of the precision used, the Streaming FPS solver of Tesla C2050 was slightly slower than the non-streaming solvers on small grids, but became the fastest when $n \geq 16{,}384$. Streaming FPS was capable of solving largest grids roughly ten times faster than the CPU in single precision, and five times faster in double precision.

10.3.5 Multi-GPU Solver

Thanks to improvements in CPU-GPU memory transfers, the Streaming FPS Solver can be extended to perform computation on multiple GPUs connected to the same motherboard. Results in this subsection were obtained on two multi-GPU testbeds: the first one consisted of three Tesla C1060 cards, and the second one was an M2050 GPU computing module (same as two Tesla C2050 cards). The observed performance is presented in Figure 10.16. Only single precision results are shown for Tesla C1060 due to a significantly slower double precision arithmetic on this card.



(A) Tesla C1060 (single precision)

(B) Tesla C2050

FIGURE 10.16: The Streaming FPS solver performance on multiple GPUs.

* Two GPUs sharing the same PCI-Express port

Due to no overlapping in the streaming solver on Tesla C1060 (see the remark in the previous subsection), the observed overall performance was relatively low. Nevertheless, a two-fold speed-up was observed when two GPUs were used (Figure

10.16 A). This was only the case if two GPUs were connected to separate PCI Express ports on the motherboard. Since the first testbed had only two such ports, two graphics cards were connected to one port. When these two cards were used instead, the performance gain was only roughly 25% due to a significantly reduced CPU-GPU data transfer throughput. Nevertheless, if all the GPUs were connected to separate PCI Express ports (or if it was possible to use a more recent PCI Express revision with a higher bandwidth), then the Streaming FPS solver is likely to achieve close to linear speed-up for sufficiently large Poisson grids.

Unexpectedly, when the solver was run on three GPUs, the performance was slightly lower than on two GPUs connected to separate ports (Figure 10.16 A). This was connected with the fact that the GPU which was not sharing the PCI Express port could finish computation in roughly half the time required by the other two GPUs, effectively wasting half of its computing power. The other two GPUs had to process 67% of data using a single PCI Express port. In the two GPUs case, each card had to process 50% of data on separate PCI Express ports, thus the higher performance. More sophisticated load balancing techniques are required to improve the utilisation of computational power of such non-symmetric multi-GPU systems.

GPUs in the M2050 platform allow for a maximum level of overlap, and in consequence, the Streaming FPS solver run roughly four times faster (Figure 10.16 B). However, in the M2050 system, GPUs are also connected to the same PCI Express port. In consequence, the performance improvement on two GPUs was greatly limited — the solver ran only 10–20% faster, regardless of the arithmetic precision.

10.3.6 Comparison with the Existing Single GPU Implementation

In this subsection, the GPU solver devised in this chapter is compared with another 2D Poisson's Equation solver proposed by Bleichrodt et al. (2012). This is to confirm that the modifications proposed in Subsection 10.2.2 indeed lead to an improved performance. Bleichrodt et al. (2012) performed their experiments on a Tesla M1060, which has the same specification as the Tesla C1060. Also, they attempted to solve problems on grids up to $4{,}096 \times 4{,}096$, which are too small to benefit from streaming, therefore only the performance of the non-streaming solver is compared.

The first proposed modification changed the way the data was packed and unpacked for the DST-I operation. Bleichrodt et al. (2012) report that in their implementation this phase amount for more than 50% of the overall DST-I time. The results from profiling confirmed that our solver spent only 22–24% of time on data manipulation. The improved performance is due to the smaller amount of transferred data and better efficiency of GPU global memory access.

Bleichrodt et al. (2012) use the Poisson solver as a part of a larger framework. The authors claim that on larger grids, their Poisson solver accounts for 75% of the total execution time, therefore the times they have reported were multiplied by a factor of 0.75. The comparison is summarised in Table 10.1.

Table 10.1: Comparison of the total execution times (in milliseconds) with the 2D Poisson's Equation solver by Bleichrodt et al. (2012).

	Bleichrodt et al.		Non-streaming Fast Poisson Solver		
Grid size (n)	GPU	CPU (1 core)	GPU	CPU (1 core)	CPU (6 cores)
512	1.47	40.5	1.50	9.50	5.27
1,024	4.99	195	4.43	36.0	10.3
2,048	19.8	950	15.2	153	33.6
4,096	83.9	4,003	70.3	669	145

The approach with the 1D DST-I and the tridiagonal solver instead of the 2D DST-I method used by Bleichrodt et al. (2012), may not be better for small grids (it was slightly slower for $n = 512$), but it is definitely faster for larger grids. Their implementation is 19% slower when $n = 4{,}096$ and even 30% slower when $n = 2{,}048$.

Furthermore, the claimed speed-up of a factor 50 over the single-core CPU implementation is caused by poor CPU implementation, rather than efficient GPU code. Indeed, the reported CPU times are between 4.3 and 6.2 times longer than the ones observed for our CPU solver. Of course, an Intel Xeon L5420 2.50 GHz CPU used by Bleichrodt et al. (2012) is slower than Intel i7-980x, but a single core should not require more than 70–80% extra time to complete the same computation.

10.4 Conclusions

- This chapter presents several implementations of the Fast Solver for 2D Poisson's Equation. They are adaptations of the Fourier analysis method presented in the previous chapter, but designed for shared memory multi-core CPU and multi-GPU systems, with an emphasis on scalability, i.e. the ability to solve arbitrarily large Poisson problems.

- Our multi-threaded CPU implementation, based on the FFTW library, delivers consistently high performance on modern multi-core CPUs. For large grids the execution time is dominated by the tridiagonal solver due to increasing cache miss ratio. It is impractical to devise a data layout optimal for both DST-I and tridiagonal solver steps. Nevertheless, our CPU solver obtained over 30 GFLOPS in single precision on six-core Intel i7-980x (25 GFLOPS in double precision), and is capable of solving the Poisson problem with one billion variables in 23 seconds in single precision, or 33 seconds in double precision.

- The single GPU implementation, based on the CUFFT library, significantly outperforms the CPU version, even taking the cost of CPU-GPU communication into account. However, the maximum problem size is limited by the GPU global memory (256 million variables on Tesla C2050 in single precision). Unlike the CPU case, here the performance limiting factors are the DST-I speed, and CPU-GPU memory transfers accounting for 35–40% of the total execution time. On GTX 480, the GPU solver achieves roughly 200 GFLOPS in single precision and 75 GFLOPS in double precision.

- The proposed non-streaming GPU implementation compares favourably with a recently published 2D Poisson's Equation solver by Bleichrodt et al. (2012). The data manipulation required to calculate DST-I is reduced from 50% to roughly 23%. The other solver is up to 30% slower on Tesla C1060 GPU.

- The introduction of the *streaming processing* to our GPU solver, allows us to solve arbitrarily large Poisson problems at the cost of tripling the amount of data exchanged between the CPU and the GPU. At the same time, the Streaming FPS Solver enables concurrent computation and data transfers using multiple CUDA streams and asynchronous copy engines available on GPUs.

- Tesla C1060 and GTX 480 have only one copy engine, thus the amount of attainable overlap is limited. Indeed, even for largest grids 50% of the total execution time is spent on non-overlapped memory transfers on GTX 480. No overlapping is observed on an older Tesla C1060 device because it does not support asynchronous transfers of non-linear memory. In comparison with the streaming implementation, the non-streaming solver is at least 30% faster on GTX 480, and 45% faster on Tesla C1060. Nevertheless, thanks to the proposed modification it is possible to solve the Poisson problem with 49,152 $\times$ 49,152 points (9 GB of input data in single precision) using the GPU with only 1.5 GB memory in less than 14 seconds.

- The Streaming FPS solver delivers an outstanding performance on GPUs with two asynchronous copy engines (Tesla C2050) — up to 90% of memory transfers are overlapped with computation. In fact, the streaming solver becomes faster than the non-streaming implementation for $n \geq 16{,}384$ and performs up to ten times faster than the CPU implementation in single precision (five times in double precision). The streaming solver on Tesla C2050 is capable of solving the Poisson problem with one billion variables in roughly 3 seconds in single precision (6 seconds in double precision).

- The Streaming FPS solver can be extended to run on multiple GPUs sharing the same motherboard. The results suggest that linear speed-ups with respect to the number of GPUs is possible, provided the devices are connected to separate PCI Express ports. If that is not the case, performance is degraded due to lower CPU-GPU bandwidth, and thus lower data transfer rates.

Chapter 11

Multi-GPU Solver for 3D Poisson's Equation

This chapter presents the FFT-based solver extended to accommodate the 3D Poisson's Equation. The 3D FPS solver follows the same approach as in the previous chapter, although requires a multi-dimensional Discrete Sine Transform and considerable implementation adjustments due to the different data layout and access patterns. As in the previous chapter, the main objective was to maximise the scalability, i.e. ability to solve arbitrarily large problems in acceptable time.

This chapter is organised as follows. In Section 11.1 the FPS solver for 3D Poisson's Equation is derived, and the related literature is revised. The details on several CPU- and GPU-based implementations proposed in this chapter are given in Section 11.2. The results of numerical experiments are presented and discussed in Section 11.3. Finally, the conclusions are given in Section 11.4.

11.1 Shared Memory 3D Fast Poisson Solver

In this section the method described in Section 10.1 is extended to accommodate the 3D Poisson's Equation. The related work is revised in Subsection 11.1.1. The mathematical background behind the 3D solver is elaborated in Subsection 11.1.2.

11.1.1 Related Work

Massively parallel solvers for 3D Poisson's Equation based on Fourier analysis have been considered for at least 15 years. In 1999, Giraud et al. proposed an MPI implementation and applied it to atmospheric simulation. Their solver performed the 2D FFT separately in each direction, followed by a solution of a number of tridiagonal systems. In this chapter, the same variant of the FFT-based Poisson solver is considered. Giraud et al. (1999) reported good weak scalability (increasing the number of processing elements, but keeping the amount of data per PE constant) — speed-up by a factor of 71.35 was reported on the 128-node Cray T3E system.

Serafini et al. (2005) have also focused on the weak scalability of the 3D FPS solver, but they combined it with several other methods: James Algorithm, Method of Local Corrections and Adaptive Mesh Refinement. The authors presented results of experiments on IBM SP Seaborg system (peak performance of 10 TFLOPS). The solver spent only 5–9% of the total execution time on communication and was able to solve 3D Poisson problems on grids of up to $2{,}560^3$ points (roughly 64 GB of data in single precision) in less than a minute on 4,096 processors.

An alternative approach to the solution of 3D Poisson's Equation is to perform a transformation in all three dimensions, and then scale the result by the corresponding eigenvalues. This approach was followed by Wu et al. (2014) — the authors report an outstanding performance of up to 375 GFLOPS on the GTX 480 (single precision). The same paper investigates the performance of an FPS variant with 2D FFT and a tridiagonal solver. The latter approach is slower, but it is applicable to a wider range of boundary conditions (Neumann, periodic). A single precision performance of up to 260 GFLOPS was reported on GTX 480.

The performance of any 3D FPS depends mostly on a multi-dimensional FFT implementation. The solvers of Wu et al. (2014) employ a hand-tuned GPU implementation of the 3D Fast Fourier Transform proposed by Wu and JaJa (2012), that is significantly faster than the CUFFT library (NVIDIA, 2012b), and the Nukada FFT library (NUFFT; Nukada and Matsuoka, 2009).

A major drawback of the optimisations used by Wu and JaJa (2012) is that they require that the entire grid is stored in the relatively small device memory, effectively limiting the maximum problem size. In consequence, the largest grids that can

be solved without running out of memory consist of 2^{25} points (128 MB in single precision) on the GTX 480, and 2^{27} points (512 MB) on the Tesla C2050 (Wu et al., 2014). This is far from the global memory size on these devices: 1,536 MB and 3,072 MB, respectively. Furthermore, multi-GPU processing is impractical due to the requirement that all the data must reside in the GPU memory.

11.1.2 Method Derivation

The general form of 3D Poisson's Equation is similar to the 2D version, adding only a term for the third dimension (cf. Subsection 9.2.1) and is defined as

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = f \tag{11.1}$$

on some domain Ω. In this study, cuboid grids are considered, i.e.

$$\Omega = \{(x, y, z) : x_{min} < x < x_{max},\ y_{min} < y < y_{max},\ z_{min} < z < z_{max}\}. \tag{11.2}$$

Using the FDM method on an $n \times m \times l$ grid of points uniformly distributed in each dimension, Equation (11.1) on domain Ω can be discretised to a set of equations

$$\begin{aligned} h_x^{-2}\left(v_{i-1,j,k} - 2v_{i,j,k} + v_{i+1,j,k}\right) + h_y^{-2}\left(v_{i,j-1,k} - 2v_{i,j,k} + v_{i,j+1,k}\right) + \\ + h_z^{-2}\left(v_{i,j,k-1} - 2v_{i,j,k} + v_{i,j,k+1}\right) = f_{i,j,k}, \end{aligned} \tag{11.3}$$

assuming:

$$h_x = \frac{x_{max} - x_{min}}{n+1}, \quad h_y = \frac{y_{max} - y_{min}}{m+1}, \quad h_z = \frac{z_{max} - z_{min}}{l+1} \quad \text{and}$$
$$v_{i,j,k} \approx \phi(x_i, y_j, z_k), \quad f_{i,j,k} = f(x_i, y_j, z_k),$$
$$\text{where } x_i = x_{min} + ih_x, \quad y_j = y_{min} + jh_y, \quad z_k = z_{min} + kh_z.$$

Similar to the 2D FPS solver (cf. Chapter 10) in this chapter the DST-I version of Discrete Sine Transform is considered (Press et al., 2007, §12.4.1). In the 3D case, the 2D version of the transformation has to be used. The 2D DST is simply a

concatenation of two 1D DST-I operations along the different dimensions:

$$\begin{aligned} v_{i,j,k} &= \sum_{J=1}^{m} \hat{v}_{i,J,k} \sin(jJ\varphi_m) \\ &= \sum_{I=1}^{n} \left(\sum_{J=1}^{m} \hat{v}_{I,J,k} \sin(jJ\varphi_m) \right) \sin(iI\varphi_n) \\ &= \sum_{I=1}^{n} \sum_{J=1}^{m} \hat{v}_{I,J,k} \sin(iI\varphi_n) \sin(jJ\varphi_m), \end{aligned} \tag{11.4}$$

$$f_{i,j,k} = \sum_{I=1}^{n} \sum_{J=1}^{m} \hat{f}_{I,J,k} \sin(iI\varphi_n) \sin(jJ\varphi_m), \tag{11.5}$$

where $\varphi_n = \frac{\pi}{n+1}$ and $\varphi_m = \frac{\pi}{m+1}$.

When $v_{i,j,k}$ values are substituted according to Equation (11.4), Equations (11.3) can be rewritten as

$$\begin{aligned} &h_x^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} \hat{v}_{I,J,k} \sin(jJ\varphi_m) \left[\sin((i-1)I\varphi_n) - 2\sin(iI\varphi_n) + \sin((i+1)I\varphi_n) \right] + \\ &+ h_y^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} \hat{v}_{I,J,k} \sin(iI\varphi_n) \left[\sin((j-1)J\varphi_m) - 2\sin(jJ\varphi_m) + \sin((j+1)J\varphi_m) \right] + \\ &+ h_z^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} (\hat{v}_{I,J,k-1} - 2\hat{v}_{I,J,k} + \hat{v}_{I,J,k+1}) \sin(iI\varphi_n) \sin(jJ\varphi_m) = f_{i,j,k}. \end{aligned} \tag{11.6}$$

The expressions in the square brackets in Equations (11.6) can be simplified with Equation (9.7): $\sin(A-B) - 2\sin A + \sin(A+B) = -4\sin A \sin^2 \frac{B}{2}$. This allows us to rewrite Equations (11.6) as

$$\begin{aligned} &h_x^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} -4\hat{v}_{I,J,k} \left(\sin^2 \frac{I\varphi_n}{2} \right) \sin(iI\varphi_n) \sin(jJ\varphi_m) + \\ &+ h_y^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} -4\hat{v}_{I,J,k} \left(\sin^2 \frac{J\varphi_m}{2} \right) \sin(iI\varphi_n) \sin(jJ\varphi_m) + \\ &+ h_z^{-2} \sum_{I=1}^{n} \sum_{J=1}^{m} (\hat{v}_{I,J,k-1} - 2\hat{v}_{I,J,k} + \hat{v}_{I,J,k+1}) \sin(iI\varphi_n) \sin(jJ\varphi_m) = \\ &= \sum_{I=1}^{n} \sum_{J=1}^{m} \hat{f}_{I,J,k} \sin(iI\varphi_n) \sin(jJ\varphi_m). \end{aligned} \tag{11.7}$$

A matrix of elements of the form $\sin(ij\varphi)$ is non-singular (Kincaid and Cheney, 2002, §9.9), thus the problem can be reduced to the solution of the following set of equations:

$$h_z^{-2}\hat{v}_{i,j,k-1} - \left(4h_x^{-2}\sin^2\frac{i\varphi_n}{2} + 4h_y^{-2}\sin^2\frac{j\varphi_m}{2} + 2h_z^{-2}\right)\hat{v}_{i,j,k} + h_z^{-2}\hat{v}_{i,j,k+1} = \hat{f}_{i,j,k}. \tag{11.8}$$

For fixed i and j, this becomes a tridiagonal system with l unknowns. Since $1 \leq i \leq n$ and $1 \leq j \leq m$, there are nm such systems to be solved.

To calculate the $\hat{f}_{i,j,k}$ values, the inverse 2D DST-I has to be performed. This is a forward 2D DST-I operation multiplied by a scalar factor (cf. Subsection 10.2.2).

11.2 Implementation Details

This section describes how the method derived in Subsection 11.1.2 translates into an algorithm and how it was implemented. First, the details of how the 2D DST-I is computed on the CPU and the GPU are given in Subsection 11.2.1. The Fast Solver for 3D Poisson's Equation (3D FPS) is outlined in Subsection 11.2.2. Finally, performance considerations are addressed in Subsection 11.2.3.

11.2.1 Computing the 2D DST-I

As was shown in Equation (11.4), the 2D DST-I can be computed as two separate 1D DST-I operations along different directions. Alternatively, the general 2D Fast Fourier Transform can be used instead. In this case, data manipulation similar to that described in Subsection 10.2.2 is required for each dimension. The size of the transformation is increased roughly two-fold in each direction, i.e. the 2D FFT would have to be computed on four times larger data.

In addition, the CPU FFT implementation (FFTW3) supports multi-dimensional real-to-real transforms, the 2D DST-I among them. The initial experiments showed, that on a single core it is up to 10% faster than two 1D DST-I calls. However, this approach experiences lower parallel efficiency when performing multiple such transformations in separate threads, e.g. in the case of a 3D FPS.

The present version of the CUFFT library (FFT implementation on the GPUs) does not support real-to-real transforms, therefore all data manipulation had to be done with explicitly implemented CUDA kernels. A 2D FFT is up to four times faster than two separate 1D FFT calls. However, the time required for additional data manipulation (cf. Subsection 10.2.2) effectively decreases the performance. Furthermore, two times more memory is required to use the 2D FFT, which is a significant disadvantage since the size of GPU memory is typically relatively small. Finally, the separation of FFT directions enables finer data granularity in streaming and multi-GPU FPS solvers — in effect, their performance can be noticeably improved.

11.2.2 The Algorithm

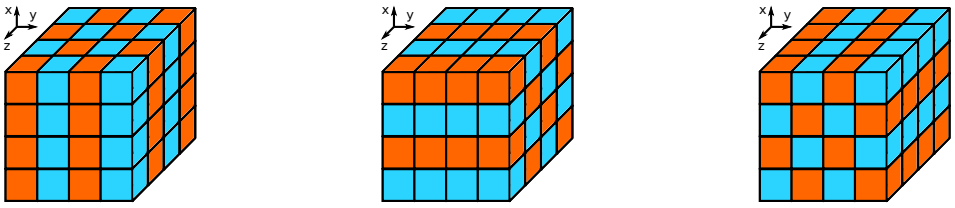
The method derived in subsection 11.1.2 and the analysis provided in this section, led to the solver outlined in Algorithm 11.1. It is similar to the 2D formulation (cf. Algorithm 9.1), but introduces an additional DST-I step along the Y-axis.

Algorithm 11.1 Fast Solver for 3D Poisson's Equation

1: Calculate DST-I along the X-axis { ml vectors of size n }
2: Calculate DST-I along the Y-axis { nl vectors of size m }
3: { At this point $\hat{f}_{i,j,k}$ values are computed. }
4: Calculate $\hat{v}_{i,j,k}$ values { Solve mn tridiagonal systems of size $l \times l$ }
5: Calculate the inverse DST-I along the Y-axis { nl vectors of size m }
6: Calculate the inverse DST-I along the X-axis { ml vectors of size n }
7: { At this point the final result ($v_{i,j,k}$ values) is computed. }

If the grid is too large to fit into GPU global memory, only the streaming implementation is capable of solving the Poisson problem. In this case, the data is divided into smaller chunks and then processed one by one on the device. This approach not only improves the scalability of the solver, but also enables multi-GPU processing.

Each streaming algorithm step requires uploading the whole grid in chunks to the GPU, processing, and then downloading the chunks back from the GPU. As discussed in Subsection 10.2.3, the computation and data transfers can overlap. However, data chunks are different for each phase (Figure 11.1). In consequence, the whole grid has to be transferred between the CPU and the GPU five times.



(A) DST-I (X-axis) (B) DST-I (Y-axis) (C) Tridiagonal solver

FIGURE 11.1: Data chunks processed in each step of the 3D Fast Poisson Solver.

11.2.3 Performance Improvements

As was shown in the two previous chapters, data should be arranged in a way that maximises the DST-I performance, possibly at the cost of the lower tridiagonal solver speed. In the case of Algorithm 11.1, the DST-I is calculated on both rows and columns, therefore both row-major and column-major data layouts could be used. In this study, column-major order was used (Figure 11.2). In order to maximise the performance on the GPU, each column has to be properly aligned (NVIDIA, 2011, §5.3.2.1). This data layout maximises the performance of the DST-I operation along the X-axis, but may result in slower execution of DST-I along the Y-axis.

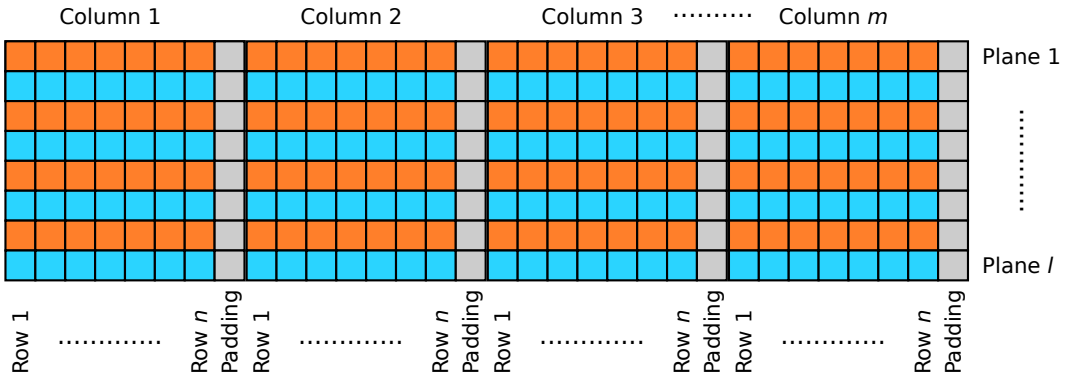


FIGURE 11.2: Data layout in the 3D Fast Poisson Solver. Properly aligned (pitch) column-major storage maximises the performance on the GPU.

The decision on a particular data layout defines the stride between consecutive elements and distances between the first elements of rows, columns, and planes. The first elements of two consecutive columns are always *pitch* (number of rows + the padding) elements apart, then the consecutive column elements are stored

contiguously. The stride between row elements is always equal to *pitch*, but the distance between the first elements of two consecutive rows may vary: if they are both in the same plane, the distance is only one element, but if they are on different planes the distance is proportional to the number of elements in each plane ($n \times m$). Finally, the first elements of vectors orthogonal to planes are located on the first plane, i.e. distance between them is always one element, and the stride is always equal to $pitch \times m$ (the distance between the first elements of two consecutive planes).

The analysis of strides and distances between the elements is important for efficient implementation of each solver step. When computing DST-I along the X-axis, the data can be treated as a stream of consecutive vectors to be transformed. In consequence, this operation can be dispatched in a single call in the streaming solver, effectively maximising the benefits from overlapping and multi-GPU processing.

Also in the case of the tridiagonal solver, data can be treated as a uniform stream, this time starting with the first elements of all systems, then the second elements, etc. This is especially beneficial on the GPUs, where hundreds of systems are solved at the same time and threads can benefit from fully coalesced global memory accesses. Furthermore, this step can also be dispatched in a single call in the streaming solver.

Multiple CUDA kernel dispatch (one for each grid plane) is required when computing the DST-I along the Y-axis, due to an inevitable inconsistency in distances between consecutive rows. The need to perform multiple dispatches may have a significant impact on the streaming implementation performance, but does not affect the non-streaming solver since the processing step is separated from the memory transfers.

The first problem is that each dispatch may consist of the relatively small number of independent vectors to transform — indeed, in most cases this number is smaller than the DST batch size. This issue can be solved by assigning work from each dispatch to a different CUDA stream, e.g. using the Round-Robin method. Otherwise, all the vectors would be assigned to the same stream and no overlap would occur.

The second problem arises on devices with only one asynchronous copy engine. As discussed in Subsection 10.2.3, to achieve any computation-communication overlap, CUDA calls have to be dispatched in groups. This cannot be done unless the number of vectors to transform in each dispatch is larger than the DST batch size. On the other hand, reducing the size of the DST batch results in lower GPU utilisation, and in consequence, reduced overall performance.

11.3 Numerical Experiments

Several implementations of FPS solvers for 3D Poisson's Equation were compared using performance models described in Subsection 11.3.1: the multi-threaded CPU solver (Subsection 11.3.2), FPS on a single GPU (Subsection 11.3.3), the Streaming FPS solver (Subsection 11.3.4), and the multi-GPU solver (Subsection 11.3.5). Moreover, the comparison with other existing 3D Poisson solvers is presented in Subsections 11.3.6 (CPU implementations) and 11.3.7 (GPU implementations).

Technical details of two CPU models (Intel i7-980x and Xeon E5462) and three GPU models (GTX 480, Tesla C1060 and C2050) used in the experiments are summarised in Tables B.1 and B.2, respectively. All machines were equipped with 12 GB of main memory, were running the Ubuntu 12.04 Server (64-bit) operating system, and were using CUDA driver 304.88. The CPU code was compiled with `gcc` 4.6.3, and the GPU code was compiled with `nvcc` release 5.0, V0.2.1221 and CUDA Toolkit 5.0.35. Unless specified otherwise, all experiments were run in single and double floating-point arithmetic, were repeated ten times, results were averaged, and the ratio of the standard deviation to the average was confirmed to be less than 5%.

11.3.1 Performance Models

To solve the 3D Poisson's Equation, the Discrete Sine Transform has to be performed in two dimensions. In comparison to the 2D case, to solve the problem with the same number of unknowns, the 3D solver requires roughly 33% more floating-point operations. This is captured in formulae (the same as in the work of Wu et al., 2014) defining the performance model that ensures fair comparison between algorithms.

$$F_{dst(x)}(n,m,l) = \frac{ml \cdot 5n\log_2 n}{t_{dst(x)} \cdot 10^9} \qquad [GFLOPS] \qquad (11.9)$$

$$F_{dst(y)}(n,m,l) = \frac{nl \cdot 5m\log_2 m}{t_{dst(y)} \cdot 10^9} \qquad [GFLOPS] \qquad (11.10)$$

$$F_{tri}(n,m,l) = \frac{nm \cdot 8l}{t_{tri} \cdot 10^9} \qquad [GFLOPS] \qquad (11.11)$$

$$F_{overall}(n,m,l) = \frac{nml \cdot (10\log_2 n + 10\log_2 m + 8)}{t_{overall} \cdot 10^9} \qquad [GFLOPS] \qquad (11.12)$$

Here, n, m, and l denote the number of rows, columns, and planes in the grid.

11.3.2 Multi-Threaded CPU Solver

The parallel efficiency of the multi-threaded DST-I depended on the problem size, but also on the direction in which the transform was computed (Figure 11.3). In all cases, the efficiency increased with the problem size, but the differences were much more prominent in DST-I along the Y-axis. This was connected with the different dispatch method incurring additional overhead that was significant on small grids. Indeed, the observed parallel efficiency was in the range of 10–40% for small grids.

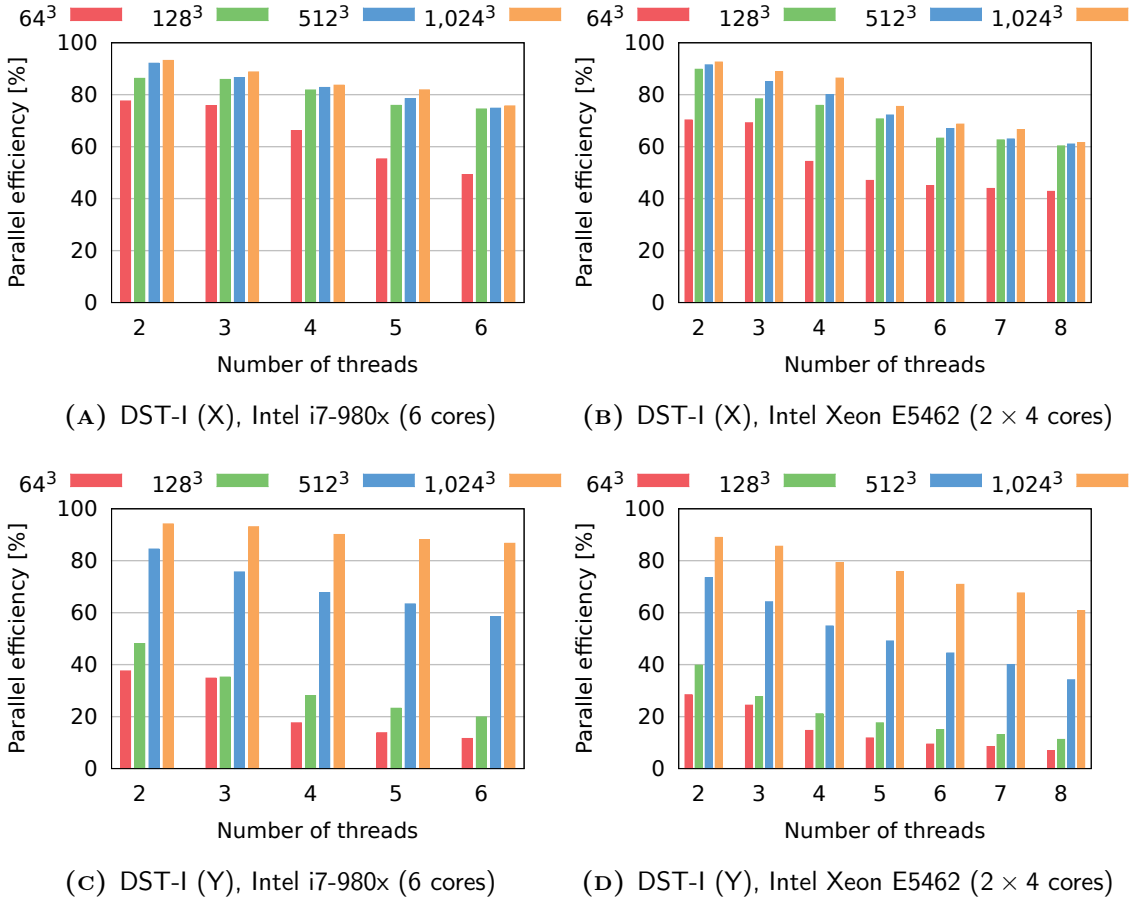
The parallel efficiency of the DST-I along the X-axis decreased slightly with the increasing number of threads, but remained at a high level even for small grids (127^3 points). When all the cores were used, efficiencies above 75% (Intel i7-980x, Figure 11.3 A), and above 60% (Xeon E5462, Figure 11.3 B) were observed.

In the case of the DST-I along the Y-axis, good parallel performance was only observed for the large grids ($1{,}023^3$ points). On Intel Xeon E5462 efficiency comparable to that of DST-I along the X-axis was observed (Figure 11.3 D). Unexpectedly, on Intel i7-980x the parallel performance was consistently above 90% (Figure 11.3 C), i.e. significantly better than in the DST-I along the X-axis. This could be connected with the high L2 cache hit ratio, which is absent on Intel Xeon E5462.

In comparison to DST-I in the 2D FPS solver (Figure 10.5), parallel efficiencies in the 3D solver were lower. This was connected with a larger number of smaller DST-I operations leading to higher parallelisation overheads. When computing DST-I along the Y-axis the overhead was even higher due to a different dispatch method.

Similar to 2D Poisson's Equation, the tridiagonal solver works on strided data. However, in the 3D case there is a greater number of smaller systems to solve which makes efficient parallelisation easier. Indeed, high parallel efficiency was observed, regardless of the CPU, problem size, and the number of threads (Figure 11.4).

On Intel i7-980x changing the grid size and the number of threads resulted in small variations in parallel performance (Figure 11.4 A). The efficiency remained at a high, 95% level. Different trends could be observed on Intel Xeon E5462 — in this case, speed-ups decreased noticeably with the problem size and the increasing number of threads (Figure 11.4 B). The parallel performance observed for small grids was significantly lower than on Intel i7-980x, but for large grids it was close to linear



(A) DST-I (X), Intel i7-980x (6 cores)

(B) DST-I (X), Intel Xeon E5462 (2×4 cores)

(C) DST-I (Y), Intel i7-980x (6 cores)

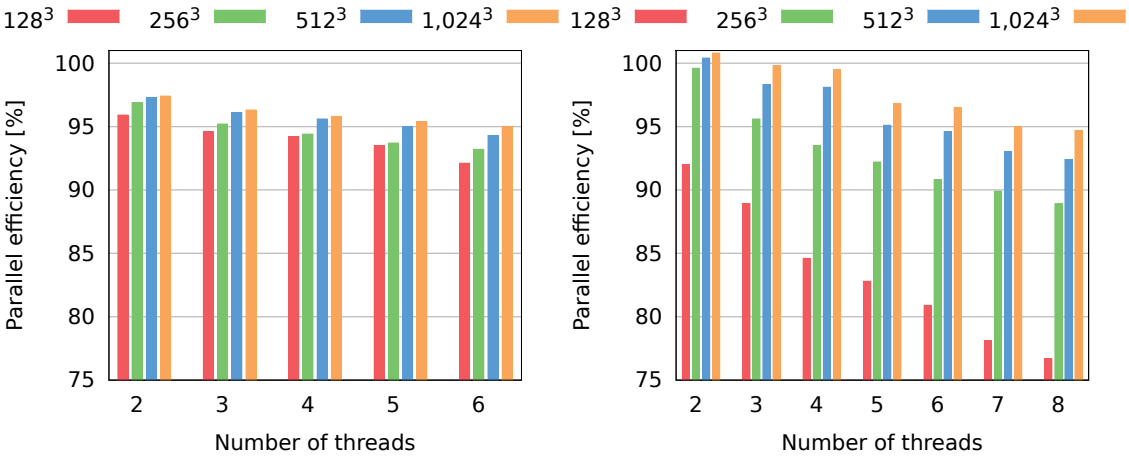
(D) DST-I (Y), Intel Xeon E5462 (2×4 cores)

Figure 11.3: The parallel efficiency of the multi-threaded DST-I (single precision). The efficiency is lower for smaller grids, especially in DST-I along the Y-axis, due to higher parallelisation overheads caused by a different dispatch method.

speed-up on up to 4 cores. Once the second quad-core unit was used, the efficiency dropped, but remained at a high, 95% level on all eight cores of Intel Xeon E5462.

In comparison to the tridiagonal solver in the 2D case (cf. Figure 10.6), parallel efficiencies were significantly higher, especially for large grids — in the 3D case, considerably smaller systems were solved, which led to a higher cache hit ratio.

The Discrete Sine Transform along the X-axis was the fastest step (Figure 11.5). The speed increased with the number of rows that enabled faster FFT computation. For large grids, the DST-I along the Y-axis was roughly 2–3 times slower due to less favourable data layout (longer stride between the consecutive vector elements).



(A) Intel i7-980x (6 cores) (B) Intel Xeon E5462 (2 × 4 cores)

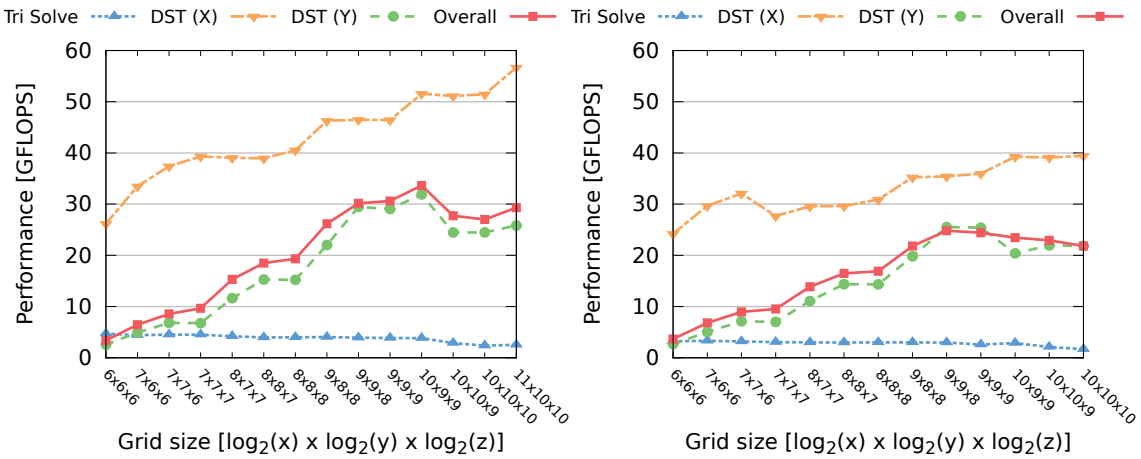
FIGURE 11.4: The parallel efficiency of the multi-threaded tridiagonal solver (single precision). Different cache configurations on the CPUs led to the contrasting performance.

The speed of DST-I along the Y-axis increased with the number of rows (more vectors to transform per dispatch, thus better parallel performance), but also with the number of columns (longer vectors for FFT computation). The performance of the tridiagonal solver was constant, except for the slight decrease for the large grids due to the decreasing cache hit ratio.

The overall performance of the multi-threaded 3D solver increased with the problem size and reached the peak performance for the input data size of around 1 GB: 33.6 GFLOPS in single precision (Figure 11.5A), and 24.8 GFLOPS in double precision (Figure 11.5B). This is comparable to the peak performance in the 2D case (Figure 10.7), but in the latter case it was only observed for small grids — for large grids, the 3D solver offered better performance.

11.3.3 Single GPU Solver

In the 2D Fast Poisson Solver on the GPU, a significant amount of auxiliary memory was required, limiting the maximum batch size for DST-I and tridiagonal solver steps. In the 3D case, the vectors transformed, or tridiagonal systems solved, are much smaller but more numerous. Therefore, the memory requirements are smaller,



(A) Single precision (B) Double precision

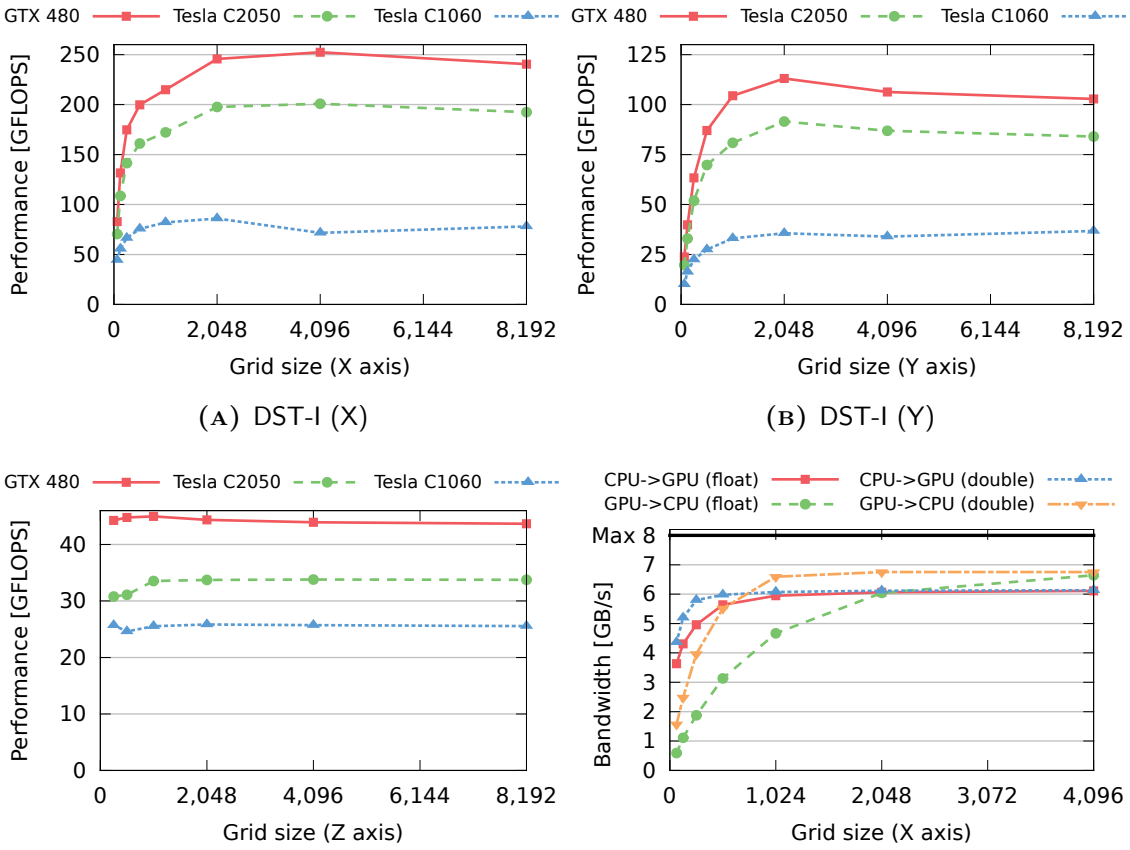
FIGURE 11.5: The performance of the multi-threaded 3D FPS Solver on six cores of Intel i7-980x. DST-I along the X-axis runs at a speed close to the maximum measured performance of the CPU. The DST-I along the Y-axis is slower due to less favourable data layout. The tridiagonal solver is the slowest operation since it is memory-bound.

and the batch size for both operations was set to 8,192. As was shown in Subsection 10.3.3, using bigger batches does not lead to any improvement in performance.

The first step was to assess the speed of each solver step and how it is influenced by the grid size. For this purpose oblong grids were used: only one dimension was changed while the other two dimensions were set to 128×64. This setup ensured that the GPU could be fully utilised. The results are summarised in Figure 11.6.

Regardless of the device and the transform direction, the performance of the Discrete Sine Transform increased quickly with the vector length to reach the peak performance for $n \geq 2{,}048$ elements (in single precision): 250 GFLOPS on GTX 480 for transform along the X-axis (Figure 11.6 A), and 113 GFLOPS for transform along the Y-axis (Figure 11.6 B). The performance on Tesla C2050 (same Compute Capability) was proportional to the theoretical peak performance. Tesla C1060 has lower Compute Capability (no global memory caching, smaller register count and shared memory size) — in this case the performance was 25% lower than expected.

DST along the Y-axis was over two times slower than the transformation along the X-axis. This was caused by less favourable data layout, effectively limiting the global



(A) DST-I (X)

(B) DST-I (Y)

(C) Tridiagonal solver

(D) PCI-Express communication

FIGURE 11.6: Performance of the 3D Fast Poisson Solver steps on a single GPU (single precision). DST-I along the X-axis is roughly two times faster than along the Y-axis, due to more favourable data layout. The DST-I speed is proportional to the theoretical peak performance; slightly slower on Tesla C1060 due to lower Compute Capability. The performance of the tridiagonal solver is proportional to the peak global memory bandwidth, since this step is memory-bound. The size of the leading dimension (n) affects the attained PCI Express bandwidth — the maximum was only observed when $n \geq 1{,}024$.

memory access coalescing in the packing step required before computing the general FFT (cf. Subsection 10.2.2). In consequence, the time spent on data manipulation was three times longer, but the time required for the FFT remained the same.

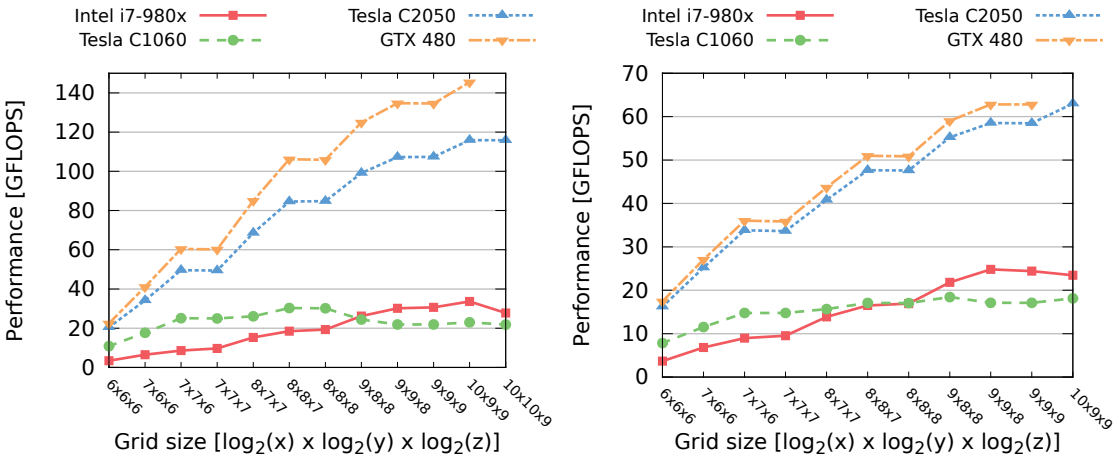
The speed of the tridiagonal solver was constant, regardless of the system size (Figure 11.6C). The observed performance was similar to that observed in the 2D solver (Figure 10.9), and was proportional to the theoretical peak global memory bandwidth since the tridiagonal solver is memory-bound.

To obtain the maximum performance on the GPU, the data has to be properly aligned (NVIDIA, 2011, §5.3.2.1). When the leading dimension, i.e. the number of rows in column-major storage, is relatively small, this may cause significant overhead when transferring data between the CPU and the GPU. Indeed, the maximum upload bandwidth is not obtained unless $n \geq 1{,}024$. When downloading data from the GPU, the maximum bandwidth is only obtained when $n \geq 2{,}048$ (Figure 11.6D). These thresholds are two times lower in double precision.

The next step was to compare the performance of the GPU solver with the multi-threaded code running on a high-end CPU (Intel i7-980x). Both devices with Compute Capability 2.0 are many times faster than the CPU implementation (Figure 11.7). The performance on Tesla C1060 was not much different than that on the CPU. In fact, for large grids the older GPU is slower because the performance of DST-I along the Y-axis deteriorates with the increasing grid size, due to the lack of global memory caching and a higher fraction of non-coalesced global memory reads.

GTX 480 delivered up to 150 GFLOPS overall performance and was up to seven times faster than the six-core CPU in single precision (Figure 11.7A). As in the 2D case, the performance of GTX 480 and Tesla C2050 is similar in double precision (Figure 11.7B), since the former device is a gaming card optimised for single precision arithmetic. Both GPUs achieved up to 63 GFLOPS, and were up to four times faster than Intel i7-980x.

In comparison to the 2D case (Figure 10.10), the 3D solver on a single GPU is slower due to lower DST-I performance. The first reason is that in the 3D case smaller vectors are transformed, limiting the maximum performance of the FFT. The second reason is more than two times slower performance of DST-I along the Y-axis.



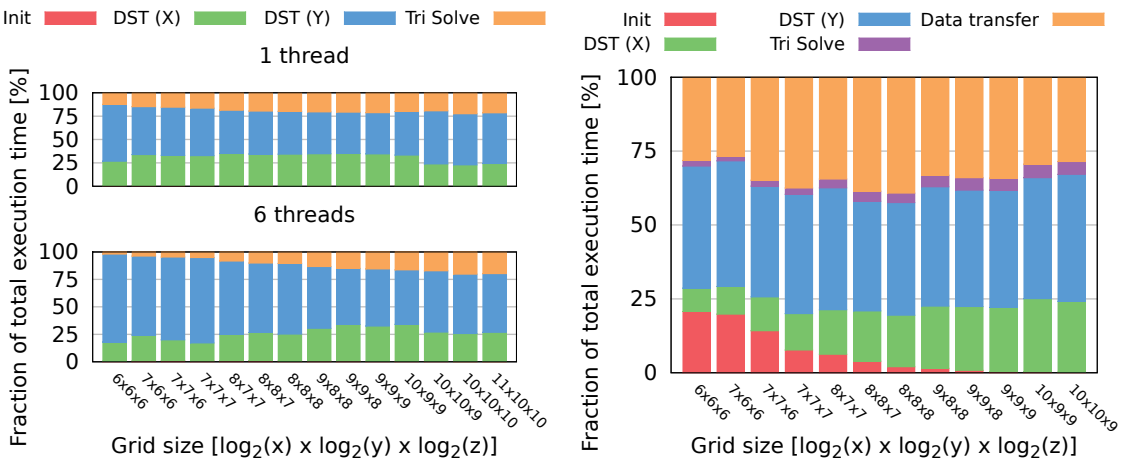
(A) Single precision (B) Double precision

FIGURE 11.7: Performance of the 3D Fast Poisson Solver on a single GPU. The speed increases with the number of rows or columns, due to faster DST-I computation. The more recent devices offer significantly better performance than the CPU.

Finally, to identify performance limiting factors, the time required for each algorithm step was measured (Figure 11.8). In the CPU case, the execution time proportions were roughly constant if only one thread was used: DST-I along the Y-axis required 50% of time, and DST-I along the X-axis and the tridiagonal solver required roughly 25% each (Figure 11.8 A). When all six cores of Intel i7-980x were used, DST-I (Y-axis) clearly dominated the solver execution. The fraction of time required for DST-I (X-axis) and the tridiagonal solver increased with the problem size and converged to proportions similar to the single thread case. This trend is connected with relatively low parallel efficiency of the transformation along the Y-axis.

These results are in contrast to the 2D solver, where the tridiagonal solve quickly became the dominant operation (Figure 10.11 A). This situation was avoided in the 3D case since the tridiagonal systems were relatively small. In consequence, the CPU cache hit ratio remained at a high level even when multiple threads were used.

DST-I along the Y-axis together with CPU-GPU data transfers were the dominant operations on the GPU (Figure 11.8 B). The fraction of time required by DST-I along the X-axis increased with the problem size, but did not exceed 25% even for the largest grids. Similar to the 2D case (Figure 10.11 B), the fraction of time required by the tridiagonal solver was small.



(A) CPU (Intel i7-980x) (B) GPU (Tesla C2050)

FIGURE 11.8: Execution time breakdown in the 3D FPS Solver (single precision). DST-I along the Y-axis takes over half of the total execution time on the CPU. This step together with the CPU-GPU communication overhead are the dominant operations on the GPU.

11.3.4 Streaming GPU Solver

The greatest shortcoming of the solution presented in the previous subsection is that the maximum problem size is bound by the relatively small GPU global memory size. Indeed, this is the limitation of most GPU implementations presented in the literature. Our Streaming FPS solver allows us to solve arbitrarily large Poisson problems, however at an additional communication cost.

In the non-streaming implementation, data has to be uploaded to, and downloaded from the GPU only once. In contrast, the streaming solver has to transfer the data for every algorithm step: three times in the case of the 2D solver, but five times for 3D Poisson's Equation. Thanks to the concurrent kernel execution and asynchronous CPU-GPU data transfer capabilities of modern GPUs, it is possible to hide some of this communication overhead. The first step was to estimate the performance gain from overlap attainable on different graphics cards.

All the experiments described in this subsection were conducted on all three devices listed in Table B.2 (Appendix B). The results on the older generation GPU (Tesla C1060) showed that no overlapping was achieved regardless of the number of CUDA streams used. Even though all kernel calls and memory transfers were performed

in an asynchronous manner with respect to the CPU, the execution on the GPU is completely serialised because devices with Compute Capability 1.3 do not support overlapping for transfers of non-linear memory, e.g. `cudaMemcpy2DAsync` (NVIDIA, 2011, §3.2.5.2). In consequence, the results for the Tesla C1060 are omitted.

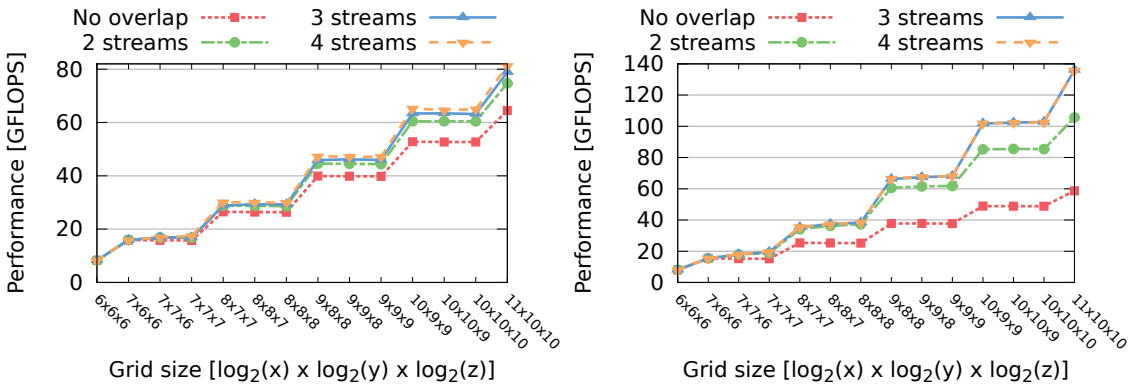
The performance of the streaming DST-I executed in different directions on two GPU cards is presented in Figure 11.9. In each case, using more than three streams did not provide any significant improvement in performance.

When DST-I was executed along the X-axis, the performance was increasing with the number of streams on both devices. The increase was relatively small on GTX 480 (Figure 11.9 A), equipped with only one asynchronous copy engine, effectively limiting the maximum attainable overlap. No such limitations were present on Tesla C2050 with two copy engines. In this case, the performance improvement from overlapping was significant and led to noticeably faster execution than on the GTX 480 (Figure 11.9 B), although the latter has roughly 30% higher peak performance.

On GTX 480, DST-I along the X-axis executed at up to 80 GFLOPS, which is roughly three times slower than the non-streaming implementation. In contrast, the streaming version on Tesla C2050 achieves up to 140 GFLOPS, which is only 15% slower compared to the non-streaming performance. Furthermore, on both GPUs the streaming DST-I run slightly slower than in the 2D solver (Figure 10.12), due to the smaller vectors transformed in the 3D case.

Due to the different dispatch method and limitations of GPUs with only one copy engine no overlapping could be observed on GTX 480 when computing DST-I along the Y-axis (Figure 11.9 C). The time required for CPU-GPU data transfers roughly doubles the time required for this operation. In comparison, the level of overlapping observed on Tesla C2050 was similar to DST-I along the X-axis despite the different dispatch method. In consequence, almost 100 GFLOPS was obtained on Tesla C2050, which is comparable to the non-streaming implementation — in this case, almost all communication overhead is effectively hidden due to overlapping.

Performance characteristics of the streaming tridiagonal solver are different than in the case of DST-I. On both GPUs, the most significant improvement could be observed when the second stream was introduced (Figure 11.10). Furthermore, using more than three CUDA streams did not results in any noticeable improvement.



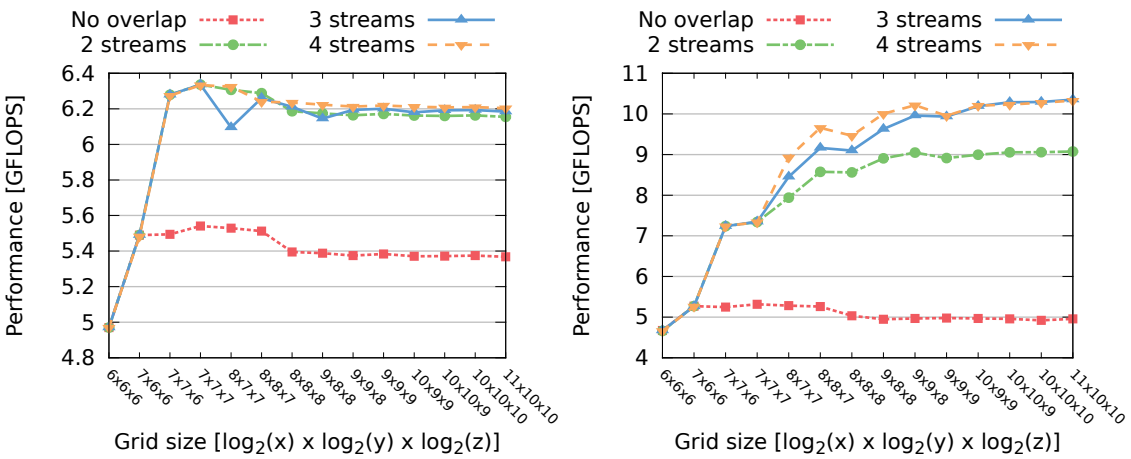
(A) DST-I (X), GTX 480 (1 copy engine)

(B) DST-I (X), Tesla C2050 (2 copy engines)

(C) DST-I (Y), GTX 480 (1 copy engine)

(D) DST-I (Y), Tesla C2050 (2 copy engines)

FIGURE 11.9: Performance of the DST-I executed on multiple CUDA streams (single precision). The results include CPU-GPU memory transfers cost. Introducing the second and the third CUDA streams leads to a significant increase in performance, especially on device with two copy engines. In the case of DST-I along the Y-axis, on GTX 480 no improvement is observed since the required grouped dispatch is not possible in this case.



(A) GTX 480 (1 copy engine)

(B) Tesla C2050 (2 copy engines)

FIGURE 11.10: Performance of the tridiagonal solver executed on multiple CUDA streams (single precision). The results include CPU-GPU memory transfers cost. The overlapping on Tesla C2050 allows for the tridiagonal solver to run over two times faster. The performance improvement on GTX 480 is smaller, since the problem is memory-bound and the data can only be transferred in one direction at a time.

Thanks to overlapping, the tridiagonal solver runs roughly 15% faster on GTX 480 (Figure 11.10A). The performance was the same regardless of the number of additional CUDA streams. A slight performance drop could be observed for three streams and $255 \times 127 \times 127$ grid, due to an uneven work distribution among the streams. In this case, 32,767 systems had to be solved in four batches of 8,192, resulting in two streams with one batch, and one stream with two batches. In consequence, there was no overlap when the last batch was processed.

On the Tesla C2050 the introduction of the third CUDA stream resulted in further performance improvement — to 10 GFLOPS, or two times faster than when there was no overlapping (Figure 11.10B). The streaming tridiagonal solver is up to seven times slower (GTX 480), or roughly three times slower (Tesla C2050) in comparison to the non-streaming implementation. Such a significant difference in performance is connected with the fact that the tridiagonal solver is memory-bound, and in consequence, more sensitive to communication overhead.

The tridiagonal solver in the 3D case was up to 25% faster than in the 2D solver (cf. Figure 10.13). Furthermore, no performance degradation was observed for

larger grids — in the 3D case, a large number of small systems is solved, effectively increasing the global memory cache hit ratio.

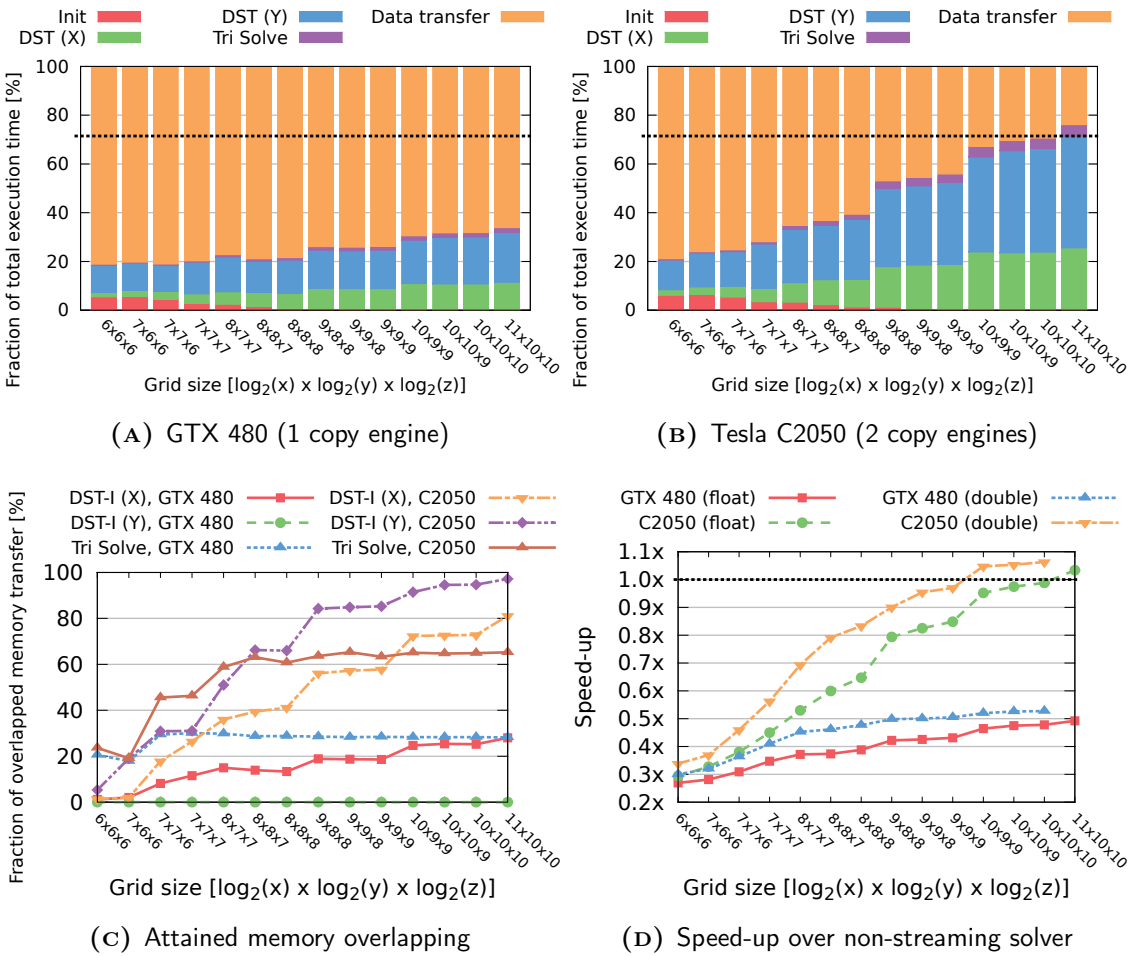
In our streaming 3D solver two times more data has to be transferred between the CPU and the GPU than in the 2D case. In consequence, the communication overhead was significantly higher and became the performance limiting factor. On small grids, up to 80% of the total execution time was spent on non-overlapped memory transfers (Figure 11.11). This fraction decreased with the increasing problem size on both devices, however the scale of improvement depended on the number of asynchronous copy engines.

On GTX 480 (one copy engine), the communication overhead accounts for more than 60% of the total time even on large grids (Figure 11.11 A). This is more than double the time spent on communication in the non-streaming solver (dotted line on the plot). Indeed, the streaming solver is over two times slower on GTX 480, regardless of the precision (Figure 11.11 D).

In contrast, when two asynchronous copy engines are available, then the fraction of time spent on non-overlapped memory transfers is decreasing with the increasing grid size (Figure 11.11 B). For the largest grids, overheads on Tesla C2050 were down to just over 20% and were lower than the CPU-GPU communication time in the non-streaming solver. Indeed, on large grids the Streaming FPS solver is faster than the non-streaming implementation (Figure 11.11 D).

As expected, the fraction of overlapped memory transfers during DST-I computation increased with the grid size (Figure 11.11 C). This was not the case when DST-I was computed along the Y-axis on GTX 480 — no overlap could be observed since in the required multiple dispatch, the CUDA calls could not be grouped. Furthermore, the amount of overlap during the solution of tridiagonal systems is limited, since the problem is memory-bound and the processing time spent on each chunk is shorter than the time required for data transfer. On Tesla C2050, almost all communication during DST-I along the Y-axis could be overlapped, over 80% during DST-I along the X-axis, and over 60% during the tridiagonal solver step.

Total execution times of all 3D Poisson's Equation solvers on the GPU increased in almost linear fashion with the input data size (Figure 11.12). The non-streaming solver was significantly faster on small grids due to lower communication overhead.



(A) GTX 480 (1 copy engine)

(B) Tesla C2050 (2 copy engines)

(C) Attained memory overlapping

(D) Speed-up over non-streaming solver

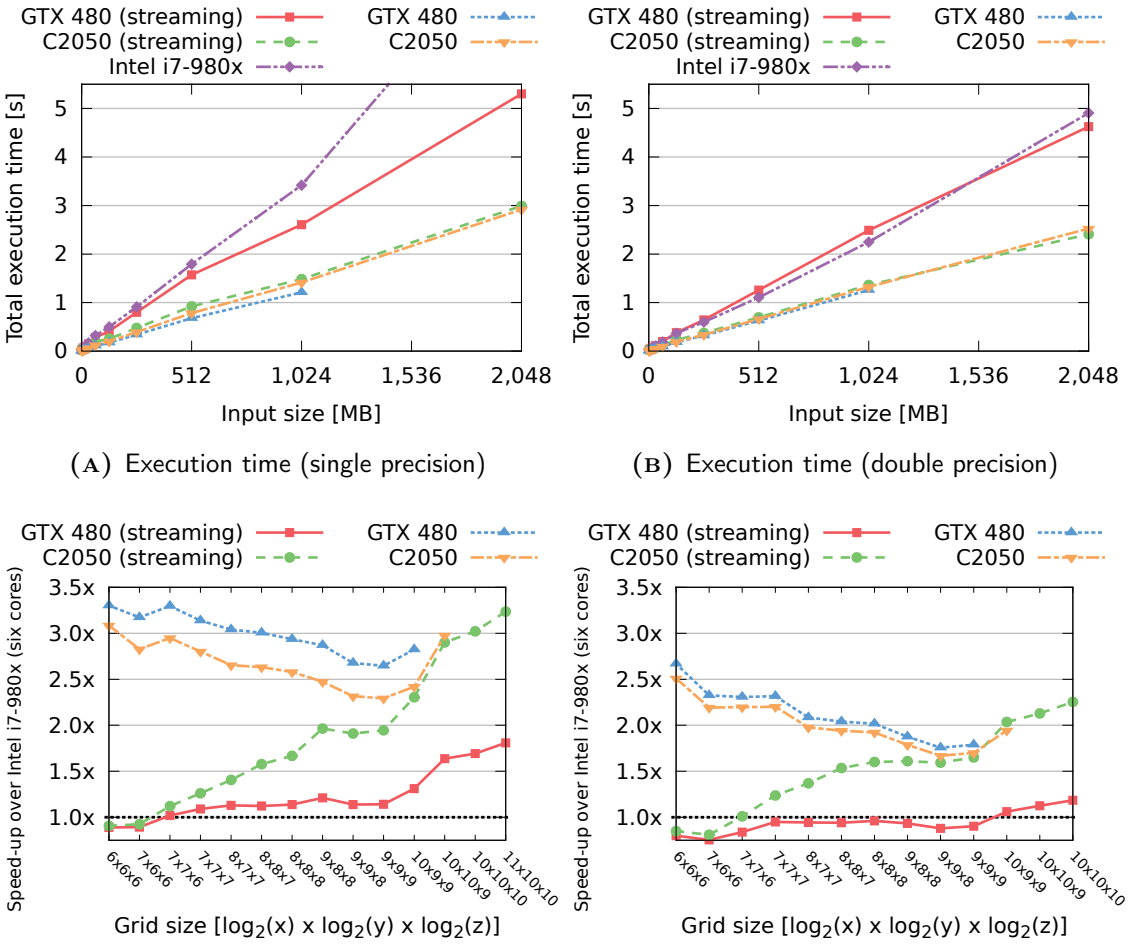
Figure 11.11: Execution time breakdown in the Streaming FPS solver and the impact of streaming on the overall performance (three CUDA streams, single precision). The dotted line denotes the level of overheads in the non-streaming solver. The CPU-GPU communication can amount up to 80% of the total execution time. The amount of overlap is significantly higher on device with two copy engines. In the DST-I case, it is possible to overlap almost all communication. The level of overlap in the tridiagonal solver is limited since the problem is memory-bound and the computation on each chunk is shorter than the time needed to transfer the data.

For large problems, the streaming solver on Tesla C2050 and non-streaming implementations required similar time to complete the computation. The performance of the streaming solver on GTX 480 was significantly lower and the execution times were closer to that of the CPU implementation.

The largest grid fitting into GTX 480 memory had 511 grid points in each dimension, and could be solved in just over a second in single precision (Figure 11.12 A). The size of problems soluble with the Streaming FPS solver is only limited by the main memory size. In this study, grids up to 8 GB were considered, i.e. up to two billion variables in single precision. To solve such an instance of 3D Poisson's Equation, Tesla C2050 required only 10.6 seconds (in comparison, the 2D solver required 8 seconds for the same number of variables). The Streaming FPS solver on GTX 480 was significantly slower due to only one asynchronous copy engine. Nevertheless, GTX 480 was capable of solving two billion variable system in less than 19 seconds. The high-end six-core CPU needed more than 35 seconds to accomplish this task.

For a grid of the same size in double precision, the execution time is almost doubled in comparison to the single precision on both GPUs — for the same input data size, the execution time is roughly 20% shorter in double precision (Figure 11.12 B). The performance cost of performing computation in higher accuracy is much smaller on the CPU. In consequence, in double precision the execution time of the streaming solver on GTX 480 is roughly the same as on Intel i7-980x. The same solver is much faster on Tesla C2050 and is capable of solving 3D Poisson's Equation on $1{,}023 \times 1{,}023 \times 1{,}023$ grid in 9.8 seconds (in comparison, the 2D solver required 6.5 seconds for the grid with the same number of variables, thus input data size).

In single precision, the non-streaming solver on both devices delivered constant performance relative to the CPU and was 2.5–3.5 times faster than the CPU implementation (Figure 11.12 C). The benefits from using the streaming processing only showed for the large grids: the solver was up to two times faster than Intel i7-980x on GTX 480, and up to 3.25 times faster on Tesla C2050. The speed-ups were less prominent in double precision: the non-streaming implementation run 2–2.5 times faster than the CPU counterpart. The Streaming FPS solver on Tesla C2050 achieved up to 2.25 times speed-up over the CPU implementation.



(A) Execution time (single precision)

(B) Execution time (double precision)

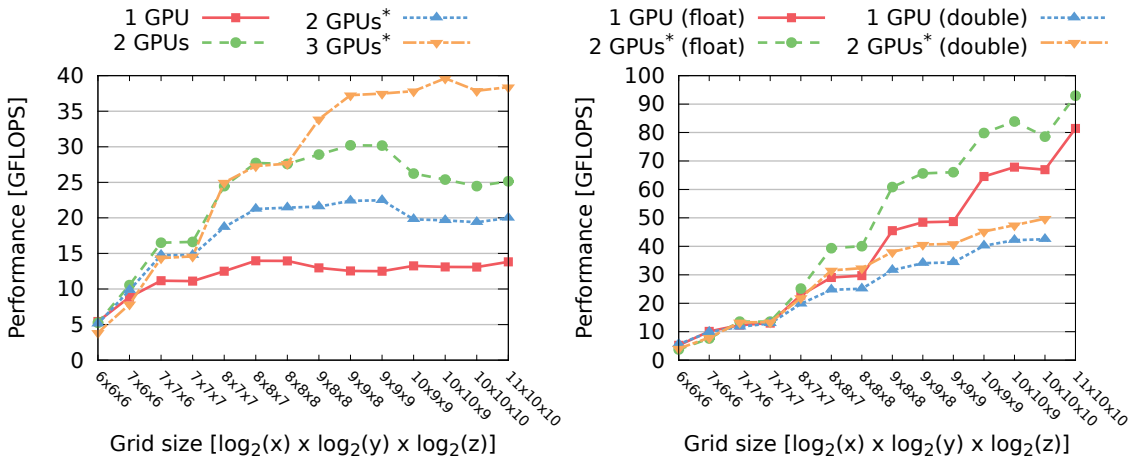
(C) Speed-up over CPU (single precision)

(D) Speed-up over CPU (double precision)

FIGURE 11.12: Comparison of the performance of 3D FPS implementations. The CPU-GPU data transfer time was taken into account. For small grids, the non-streaming implementation outperforms all other versions. As the grids get larger, the Streaming FPS solver matches the performance of the non-streaming implementation due to high level of overlapping. The Streaming FPS solver yields significantly better performance on Tesla C2050 with two asynchronous copy engines.

11.3.5 Multi-GPU Solver

Due to improvements in CPU-GPU memory transfers, the Streaming FPS solver can be extended to perform computation on multiple GPUs connected to the same motherboard. Results in this subsection were obtained on two multi-GPU testbeds: the first consisted of three Tesla C1060 cards, and the second was an M2050 GPU computing module (same as two Tesla C2050 cards). The observed performance is presented in Figure 11.13. Only single precision results are shown for Tesla C1060 due to a significantly slower double precision arithmetic on this card.



(A) Tesla C1060 (single precision) (B) Tesla C2050

FIGURE 11.13: The performance of the 3D Fast Poisson Solver on multiple GPUs.

* Two GPUs sharing the same PCI-Express port

Due to no overlapping in the streaming solver on Tesla C1060 (see the remark in the previous subsection), the observed overall performance was relatively low. Nevertheless, a two-fold speed-up was observed when two GPUs were used (Figure 11.13 A). This was only true if two GPUs were connected to separate PCI Express ports on the motherboard. Since the first testbed had only two such ports, two Tesla C1060 GPUs were connected to one port. When these two cards were used instead, the performance gain was lower due to a significantly reduced CPU-GPU data transfer bandwidth. Solving 3D Poisson's Equation is more computationally demanding than in the 2D case. In consequence, when using multiple Tesla C1060 GPUs the

performance improvement is noticeable even if the cards share PCI Express ports — the parallel efficiency on three GPUs exceeds 90% for sufficiently large grids.

GPUs in the M2050 platform allow for a maximum level of overlap (two copy engines) and in consequence the streaming FPS solver runs over five times faster (Figure 11.13B). However, in the M2050 system, GPUs are also connected to the same PCI Express port. In this case, the instruction throughput is over 65% higher than on Tesla C1060, and CPU-GPU transfers are the limiting factor again. In consequence, the solver on two GPUs is 15–35% faster, regardless of the arithmetic precision.

11.3.6 Comparison with the Existing CPU Implementation

It is difficult to compare our multi-threaded 3D FPS solver with the one proposed by Giraud et al. (1999), since the authors provided only *scaled speed-ups*, rather than absolute execution times. Another CPU implementation was proposed by Serafini et al. (2005). Their experiments were carried out on IBM SP Seaborg supercomputer equipped with 6,656 cores, with the theoretical peak performance of roughly 10 TFLOPS and the maximum measured performance of 7.3 TFLOPS, or 1.1 GFLOPS per core (classified at 4^{th} position in TOP500 Supercomputer Sites in June 2003 (`http://www.top500.org/list/2003/06/`). In comparison, the CPU used to run the multi-threaded FPS solver had six cores and the maximum measured performance of roughly 60 GFLOPS, or 10 GFLOPS per core.

Serafini et al. solved grids up to 64 GB in size. The system used in experiments had only 12 GB of main memory, therefore the performance could be compared only for the smallest reported grids (Table 11.1). Serafini et al. managed to obtain good scalability — indeed, when a number of cores was doubled their solver run two times faster. The trade-off was a low utilisation of processing capabilities of the supercomputer — the observed instruction throughput was only at 3.9% of the maximum measured performance.

In comparison, our multi-threaded 3D FPS Solver runs at roughly 45% of the peak performance, thanks to a highly optimised FFT implementation (FFTW3) and low communication overheads. In consequence, the solver running on a single six-core CPU requires less time than the implementation by Serafini et al. on 512 cores.

Table 11.1: Comparison of the total execution times and the overall performance (in GFLOPS) with the 3D FPS by Serafini et al. (2005). The number of cores is denoted by P, and the maximum attainable performance (in GFLOPS) by R_{max}. R_{act} denotes the actual performance (in GFLOPS) in accordance with the model in Equation (11.12).

					Current work			
	Serafini et al. (2005)				Single precision		Double precision	
$n = m = l$	P	R_{max}	Time [s]	R_{act}	Time [s]	R_{act}	Time [s]	R_{act}
768	128	140.5	36.7	5.34	7.7	25.2	10.1	19.4
1,024	256	281.9	43.5	11.05	17.8	27.0	22.0	21.8
1,280	512	561.8	44.5	21.74	36.5	26.5	—	—

11.3.7 Comparison with the Existing GPU Implementation

The multi-dimensional FFT used by Wu et al. (2014) is significantly faster than CUFFT transforms. In consequence, our GPU solvers are 2–3 times slower for small grids. However, optimisations used by Wu et al. severely limit the maximum soluble Poisson problem size: to 2^{25} (roughly 33.5 million) grid points on GTX 480, and 2^{27} (roughly 134 million) grid points on Tesla C2050. Our 3D Streaming FPS is capable of solving arbitrarily large Poisson problems, regardless of the GPU model. The largest grids considered in this study consisted of 2^{31} (over 2 billion) points, i.e. 64 times more than the largest grids solver by Wu et al. (2014) on GTX 480.

The performance reported by Wu et al. for the CUFFT-based version is noticeably faster than our non-streaming solver, which is also based on the CUFFT library (Figure 11.14). This difference is most likely caused by the use of different library versions: 4.2.9 and 5.0.35, respectively. Unexpectedly, the more recent library is typically slower, however it does not suffer from the significant performance drop for large grids. Furthermore, the implementation proposed in this chapter has lower memory requirements and is capable of solving grids four times larger in device memory without the need for the streaming processing.

The hand-crafted multi-dimensional FFT by Wu and JaJa (2012) offers outstanding performance. The 3D Poisson solver based on this implementation runs up to two

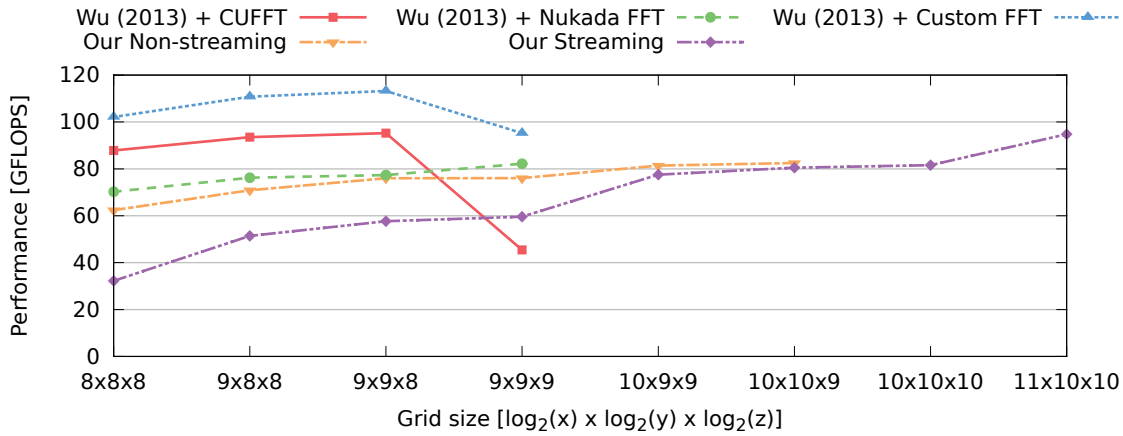


Figure 11.14: Performance comparison with the 3D Poisson solver by Wu et al. (2014) on Tesla C2050 (single precision). The CPU-GPU communication was taken into account. Solvers presented by Wu et al. yield outstanding performance, but are limited by the device memory size and can only solve relatively small problems.

times faster than the counterpart based on CUFFT. However, the maximum size of the problems is limited by the relatively small GPU memory. In contrast, our Streaming 3D FPS solver can solve arbitrarily large problems. Moreover, as the problem size increases, the overlapping becomes more effective, hence the diminishing gap between our streaming and non-streaming implementations. Furthermore, it is notable that for the largest grids our Streaming 3D FPS solver matches the performance of the fastest implementation by Wu et al. (2014) at roughly 100 GFLOPS.

11.4 Conclusions

- In this chapter, the Fast Solver for 3D Poisson's Equation is derived and several CPU and GPU implementations are presented and compared. The algorithms are designed for both multi-core CPU and multi-GPU shared memory systems with the emphasis on scalability, i.e. the ability to solve arbitrarily large Poisson problems.

- In comparison to the 2D case, the FFT-based solver requires an additional DST-I step — along the Y-axis. Due to less favourable data layout, DST-I

along the Y-axis is roughly two times slower than DST-I along the X-axis. DST-I operations in different directions can be computed separately, or they can be integrated into one step and computed using the general 2D FFT. The former approach was followed in this study, as it leads to better parallel efficiency on the CPU and enables streaming processing on the GPU.

- Our multi-threaded CPU implementation, based on the FFTW library, delivers consistently good performance on modern multi-core CPUs. The performance increased with the grid size and achieved the peak for inputs of 1 GB. Regardless of the problem size, DST-I along the Y-axis was the most time-consuming step. These results are in contrast with the 2D solver, where the peak performance was observed for small grids, because the tridiagonal solver was a dominant operation. The CPU implementation obtained 33.6 GFLOPS in single precision on a Intel i7-980x CPU (24.8 GFLOPS in double precision), and it is capable of solving 3D Poisson's Equation instances with one billion variables in 18 seconds in single precision, and 22 seconds in double precision.

- The single GPU implementation, based on CUFFT library, significantly outperforms the CPU version even taking CPU-GPU communication into account. However, the maximum problem size is limited by the relatively small GPU global memory (512 million variables on Tesla C2050 in single precision). Similar to the CPU case, the performance limiting factors are DST-I (Y-axis) and CPU-GPU memory transfers, each accounting for 30–40% of the total execution time. On GTX 480, the GPU solver achieves roughly 140 GFLOPS in single precision, and 65 GFLOPS in double precision.

- The introduction of streaming processing to the GPU solver allows us to solve arbitrarily large Poisson problems at the cost of five times larger amount of data exchanged between the CPU and the GPU. However, our Streaming 3D Fast Poisson Solver enables concurrent computation and data transfers using multiple CUDA streams and asynchronous copy engines available on GPUs.

- The Tesla C1060 and GTX 480 have only one copy engine, thus the amount of attainable overlap is limited. Indeed, on GTX 480 even for the largest grids, 65% of the total execution time is spent on non-overlapped memory transfers. No overlapping is observed on older Tesla C1060 device, because

it does not support asynchronous transfers of non-linear memory. The non-streaming solver is over 50% faster on GTX 480 than the streaming version. Nevertheless, thanks to the proposed modification, our FPS implementation can solve a Poisson problem with $2{,}047 \times 1{,}023 \times 1{,}023$ grid points (8 GB of input data) using a GPU with only 1.5 GB memory in less than 19 seconds (single precision).

- Our Streaming 3D Fast Poisson Solver delivers outstanding performance on GPUs with two asynchronous copy engines (Tesla C2050) due to overlapping up to 80% of memory transfers. In fact, the streaming solver becomes faster than the non-streaming implementation for sufficiently large grids and performs up to 3.25 times faster than the CPU implementation in single precision (2.25 times in double precision). The Streaming FPS solver on Tesla C2050 is capable of solving Poisson problem with one billion variables in less than 6 seconds in single precision, and less than 10 seconds in double precision.

- Our Streaming 3D Fast Poisson Solver can be extended to run on multiple GPUs connected to the same motherboard. The results suggest that linear speed-ups with respect to the number of GPUs is possible, provided the devices are connected to separate PCI Express ports. If that is not the case, performance is degraded due to the lower CPU-GPU data transfer rates.

Chapter 12

Conclusions and Future Work

The main aim of this project was to design, develop, and analyse algorithms to solve computationally demanding problems arising from the discretisation of partial differential equations on advanced parallel architectures. Modern high-performance computing platforms typically consist of numerous multi-core CPUs and many-core GPUs arranged in various communication topologies depending on the media used for connectivity. Significant effort was made to identify possible performance bottlenecks and devise techniques allowing for efficient parallel implementations.

Poisson's Equation was used as a model problem due to its theoretical importance and the fact that it emerges in many applications, e.g. electrostatics, electromagnetism, mechanical engineering. Furthermore, the governing equations of incompressible fluid dynamics can be recast to this form. Algorithms for finding solutions to Poisson's Equation typically experience low computation-to-communication ratio, which makes the effective parallelisation particularly challenging. In addition, a range of linear systems arising from discretisation of various computational fluid dynamics and structural engineering problems were considered in Part II.

12.1 Conclusions

The first step was to understand the performance characteristics and limitations of the modern hybrid parallel platforms comprising multiple CPUs and GPUs. Results from three case studies presented in Part I provide the required information.

Performance of the Hybrid Jacobi Solver for 2D Laplace's Equation (Chapter 3) revealed a number of communication bottlenecks in multi-GPU environments. These became especially apparent, since this algorithm has low computational intensity and is inherently *memory-bound*. Comparison between several CUDA kernels following different memory utilisation strategies revealed the importance of careful data layout that allows for a significant performance boost due to coalesced memory transactions and high cache hit ratio. The performance of our fastest kernel is close to the analytical upper bound for effective memory transfer — data is processed at a rate of up to 106 GB/s on a GeForce GTX 480 graphics card.

The communication between GPUs may become the performance limiting factor, especially in distributed memory systems. The study shows that using asynchronous MPI and CPU-GPU data transfers allows us to overlap most communication with computation. In consequence, our solver achieves close to linear speed-up on multiple GPUs even when the relatively slow Gigabit Ethernet is used for interconnect. Moreover, a procedure for automatic load balancing is introduced. The method is based on an accurate performance model, which enables us to distribute the workload almost equally in *heterogeneous* multi-GPU platforms — the fastest graphics card is idle for at most 6% of time, which leaves little space for further improvement.

Replacing the Jacobi method with Successive Over-Relaxation (Chapter 4) may lead to an order of magnitude faster convergence rate. However, the SOR method involves non-trivial data dependencies and does not parallelise as easily as the Jacobi method. The typical approach is to use the Red-Black ordering, however a straightforward GPU implementation has limited performance due to non-coalesced global memory reads. In this work we propose a novel approach where red and black nodes are stored separately in contiguous memory blocks. The new data layout allowed us to increase the memory throughput on both CPU and GPU platforms by 38%. In consequence, the GPU implementation matches the performance of a much easier Jacobi method and achieves 105 GB/s throughput on a GTX 480 graphics card.

The third case-study concerned efficient implementations of more complex algorithms and data structures on the GPU. For this purpose, a novel Parallel Multi-start Tabu Search (PMTS) for Quadratic Assignment Problem (QAP) was designed and implemented (Chapter 5). Similar to the SOR solver, PMTS uses non-intuitive data layout to maximise memory throughput in the evaluation of QAP solutions.

Furthermore, we use the first GPU implementation of the tabu list data structure, that allows us to run the Tabu Search meta-heuristic entirely on the GPU, and in consequence, avoid CPU-GPU communication overheads. Our novel GPU implementation was up to 50 times faster than the optimised CPU counterpart and the quality of solutions produced by PMTS was the same, or very close to the ones reported for other state-of-the-art meta-heuristics in the literature.

In all three case studies, our GPU implementations yielded significantly better performance in comparison to CPU counterparts. In addition to time savings, GPUs can provide noticeable reduction in the cost of running computing facilities. A commodity-grade GeForce GTX 480 GPU (£350 in October 2010) offers up to 16 times higher theoretical peak instruction throughput, and 21 times higher memory throughput[1] than a high-end six-core Intel i7-980x CPU (£770 in October 2010). Depending on a numerical problem, GTX 480 is 10–50 times faster than Intel i7-980x. The power consumption of 250 and 130 Watts, respectively, imply roughly 5–25 times lower energy cost for completing the same computation.

The second part of this thesis was dedicated to creating a scalable, general-purpose algebraic solver for large, sparse linear systems that emerge from discretisation of partial differential equations (Part II). The first step was to understand the performance of a GPU implementation of a well-established linear solver — Preconditioned Conjugate Gradients (PCG). A number of GPU solutions have been published in the literature, however optimisations proposed in this work allow our solver to run at least 50% faster than the state-of-the-art CUSP library implementation (Chapter 6). Moreover, four popular preconditioners were compared in terms of their impact on time-effectiveness of PCG, i.e. how the quality of solution changes in time.

The Distributive Conjugate Gradients (DCG) method is a recently published PDE solution algorithm offering parallelisation of the PCG solver on distributed memory systems. Motivated by the DCG approach, the Distributed Block Direct Solver (DBDS) was devised and developed as a part of this project (Chapter 7). Both methods follow the Block Relaxation scheme and divide the original system into smaller subproblems that can be solved almost independently. Since matrices in

[1] In the single channel mode. Intel i7-980x can work in the triple channel mode and the theoretical peak is tripled, however a programmer has no control on how to utilise multiple channels, and in many cases the effective memory bandwidth is as low as that of a single channel configuration.

local linear systems are fixed and only the right-hand side vectors are changing between the iterations, a new idea in DBDS was to replace PCG-style iterations with a direct sparse solver. The class of problems for which the DBDS method converges was identified, an a procedure to compute a high-quality upper bound on the convergence rate was derived.

The experiments on hybrid implementations of DCG and DBDS methods revealed a number of unexpected results (Chapter 8). First, the parallel efficiency of the hybrid DCG solver decreases quickly with the increasing number of processing elements due to the need to subtract global solution vectors in each iteration — the cost of this operation is constant, regardless of the number of subdomains and does not scale. Secondly, the performance of finding a solution to a triangular system on the GPU is highly irregular and it is determined by data dependencies from the sparsity pattern, rather than the size or the number of non-zero elements in a matrix. Finally, the speed-up from mixed-precision iterative refinement was lower than expected and also depended on the structure of a linear system. In consequence, both solvers do not offer good scalability and are impractical in most applications.

However, experiments showed that subdomain overlapping may have a positive impact on the DBDS convergence rate, especially in the case of the 2D Poisson matrix. Although the high level of overlap significantly reduces the scalability of the method — the best time-effectiveness was observed on eight CPU cores — in this case the overlapped DBDS method delivered very good performance: to solve 2D Poisson's Equation on a 2,048 $\times$ 2,048 grid, DBDS required 26 seconds on eight CPU cores and was 13.5 times faster than the state-of-the-art PCG solver on a single core.

On sequential machines Multigrid Methods have the optimal $\mathcal{O}(n)$ time complexity on linear systems from discretisation of Poisson's Equation (here, n is the number of grid elements). However, Multigrid Methods are hard to parallelise, therefore Fast Poisson Solvers (FPS) based on Fourier analysis were considered in this project. The FPS approach offers a close to optimal $\mathcal{O}(n \log n)$ time complexity on sequential platforms and allows us to achieve the *optimal* complexity on parallel architectures ($\mathcal{O}(\log n)$ in the PRAM model). The research on multi-CPU and multi-GPU implementations of FPS is presented in Part III.

FPS methods are based on the Discrete Sine Transform (DST) and finding a solution to tridiagonal systems. Calculating the DST is the first *compute-bound* problem

encountered in this project. Efficient implementations can achieve up to 67% of the theoretical peak performance on graphics cards. The initial experiments highlighted the good performance of the FPS approach: on a Tesla C2050 GPU, 2D Poisson's Equation on an 8,192 × 8,192 grid (over 67 million unknowns) can be solved in 3.4 seconds in single precision and 8 seconds in double precision. However, when run on several GPUs in a distributed memory system scalability issues were encountered and identified — the communication overhead increased noticeably with the number of processing elements, and in consequence, limited their optimal number to $\mathcal{O}(\log n)$ in a naive implementation. This was improved to $\mathcal{O}(\sqrt{n})$ at the cost of a longer DST calculation. The solver proposed in Chapter 9 achieved 98% parallel efficiency on 128 CPU cores with Infiniband interconnect. The devised performance model suggest that good scalability can be achieved on GPU clusters — a performance of over 3 TFLOPS in single precision is expected on eight Tesla C2050 GPUs.

These preliminary results suggested that Fast Poisson Solvers could achieve even better performance in the presence of shared memory (Chapter 10). In this case, highly optimised FFT implementations could be used to perform the Discrete Sine Transform. Indeed, an outstanding improvement in performance was observed: on a single Tesla C2050 GPU the above-mentioned 2D Poisson problem can be solved in 200 ms in single precision and 440 ms in double precision.

Several GPU implementations of FPS solvers have been published in the literature, however they are designed to run on a single graphics card and the maximum problem size is limited by the relatively small GPU memory. To address these limitations, an innovative *streaming processing* method was introduced — the computation is performed on smaller data chunks, and can be almost completely overlapped with memory transfers by using multiple CUDA streams and asynchronous CPU-GPU data transfers. Furthermore, our novel approach enables us to run computation on multiple GPUs since the data chunks can be processed independently. The Streaming FPS solver proposed in this thesis is the first to handle problems larger than the GPU memory, and to enable multi-GPU processing. Our method is capable of solving problems with billions of unknowns in a matter of seconds: on Tesla C2050 GPU (3 GB of memory), 2D Poisson's Equation on a 32,767 × 32,767 grid can be solved in 3 seconds in single precision (4 GB of input data) and in 6 seconds in double precision (8 GB of input data).

The bandwidth of the PCI-Express interface connecting the CPU and the GPU is the main factor limiting the Streaming FPS solver performance. In many platforms that connect multiple GPUs to the same motherboard, graphics cards share a single PCI-Express port, and in consequence, the bandwidth. In this case, the increase in performance of the Streaming FPS algorithm run on multiple GPUs is limited. However, the experiments suggest that close to linear speed-up is possible if the GPUs are connected to separate PCI-Express ports.

The improved Fast Poisson Solver was extended to accommodate the 3D Poisson's Equation (Chapter 11). In this case, a multi-dimensional FFT has to be calculated. To enable streaming processing, computing the FFT in different directions was separated in our implementation. In consequence, a performance penalty was observed due to different problem structure and limited data locality when calculating the FFT in the second direction. On the other hand, a performance gain was observed when solving smaller but more numerous tridiagonal systems. The streaming processing technique was adjusted to accommodate the 3D case, and allowed us to solve the problem on arbitrarily large grids: on two Tesla C2050 GPUs, 3D Poisson's Equation on a $1{,}023 \times 1{,}023 \times 1{,}023$ grid can be solved in 6.1 seconds in single precision (4 GB of input data) and 9.7 seconds in double precision (8 GB).

12.2 Possible Directions for Further Research

The results obtained during the research in this project highlighted several interesting directions for further study:

- The DCG method and the DBDS solver experience a convergence rate degradation when the number of processing elements, and thus subdomains, is increased. In consequence, the scalability of both methods is severely limited. The degradation is connected with the underlying Additive Schwarz decomposition. Further literature review indicated that it may be possible to modify the original Schwarz method by improving *transmission conditions* that allows us to exchange more information between the subdomains. Published results indicate that the number of iterations required for convergence can be significantly reduced (Gander, 2006; Gander et al., 2007; Loisel and Szyld, 2009),

however it is not obvious how the improved transmission conditions would affect communication, and thus the performance, in hybrid parallel platforms.

- At the moment, our Fast Poisson Solver is using a generic FFT implementation on the GPU (from the CUFFT library). This is suboptimal and can be replaced with a hand-crafted multi-dimensional Discrete Sine Transform implementation that can potentially allow for up to two-fold reduction in the processing time and the memory footprint.

- The multi-dimensional FFT in the Streaming 3D FPS solver is performed by processing the two directions separately. Further study is required to establish if data chunks in the streaming processing could be adjusted to enable 2D FFT on planes of the Poisson grid. In consequence, the amount of data transferred between the CPU and the GPU could be significantly reduced.

- In this study, the FPS method was tested on Poisson's Equation with Dirichlet boundary conditions. The results published in the literature suggest that this can be extended to accommodate Neumann and periodic boundary conditions.

References

Adams, L. and J. M. Ortega. A multi-color SOR method for parallel computation. In *Proceedings of 1982 International Conference on Parallel Processing*, pages 53–56. 1982.

Akin, J. E. *Finite elements for analysis and design*. Academic Press, Inc., 1994.

Alonso, P., R. Cortina, F. Martínez-Zaldívar, and J. Ranilla. Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA. *The Journal of Supercomputing*, 58:215–225, 2011.

Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS '67*, pages 483–485. 1967.

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

Anzt, H., S. Tomov, M. Gates, J. Dongarra, and V. Heuveline. Block-asynchronous multigrid smoothers for GPU-accelerated systems. *Procedia Computer Science*, 9:7–16, 2012.

Arnoldi, W. E. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9:17–29, 1951.

Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A

view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

Asanovic, K., R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52:56–67, 2009.

Atkinson, K. and W. Han. *Elementary numerical analysis*. J. Wiley & Sons, 2004.

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks — summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165. ACM, New York, NY, USA, 1991.

Balay, S., J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013a.

Balay, S., J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2013b. Last access: 30 September 2013.

Balay, S., W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

Barrett, R., M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, USA, 2nd edition, 1994.

Baskaran, M. and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM TJ Watson Research Center, 2009.

Battiti, R. and G. Tecchiolli. The Reactive Tabu Search. *INFORMS Journal on Computing*, 6:126–140, 1994.

Becker, D. *Parallel Unstructured Solvers for Linear Partial Differential Equations*. Ph.D. thesis, Cranfield University, 2006.

Becker, D. and C. Thompson. A novel, parallel PDE solver for unstructured grids. In I. Lirkov, S. Margenov, and J. Wasniewski, editors, *Large-Scale Scientific Computing*, volume 3743 of *Lecture Notes in Computer Science*, pages 638–645. Springer, Berlin/Heidelberg, 2006.

Bell, N. and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

Bell, N. and M. Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations. `http://cusp-library.googlecode.com`, 2012.

Benzi, M. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.

Bjørstad, P. Multiplicative and additive Schwarz methods: convergence in the two-domain case. In *Domain Decomposition Methods*, pages 147–159. SIAM, Philadelphia, 1989.

Blackford, L. S., J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software (TOMS)*, 28:135–151, 2002.

Bleichrodt, F., R. H. Bisseling, and H. A. Dijkstra. Accelerating a barotropic ocean model using a GPU. *Ocean Modelling*, 41:16–21, 2012.

Bodin, F. and S. Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming*, 17:325–336, 2009.

Bolz, J., I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proceedings of ACM Transactions on Graphics*, pages 917–924. 2003.

Boman, E. G. and B. Hendrickson. Support theory for preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 25:694–717, 2003.

Bordawekar, R., U. Bondhugula, and R. Rao. Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application. Technical Report RC24982, IBM Research Division, 2010.

Brezis, H. and F. Browder. Partial differential equations in the 20th century. *Advances in Mathematics*, 135:76–144, 1998.

Briggs, W. L., V. E. Henson, and S. F. McCormick. *A Multigrid tutorial: second edition.* Society for Industrial and Applied Mathematics, Philadelphia, USA, 2000.

Brightwell, R., R. Riesen, and K. D. Underwood. Analyzing the impact of overlap, offload, and independent progress for Message Passing Interface applications. *International Journal of High Performance Computing Applications*, 19:103–117, 2005.

Brixius, N. W. and K. M. Anstreicher. The Steinberg wiring problem. Technical report, The University of Iowa, 2001.

Brown, A. R., D. Reid, A. Asenov, and J. R. Barker. The implementation and speed-up of coloured SOR methods for solving the 3D Poisson equation on an array of transputers. In *Proceedings of the international workshop on computational electronics*, pages 171–175. 1993.

Bruck, J., C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:1143–1156, 1997.

Buatois, L., G. Caumon, and B. Levy. Concurrent number cruncher: A GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24:205–223, 2009.

Burkard, R. E., S. Karisch, and F. Rendl. QAPLIB-A Quadratic Assignment Problem library. *European Journal of Operational Research*, 55:115–119, 1991.

Burkard, R. E. and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17:169–174, 1984.

Buttari, A., J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software*, 34:17:1–17:22, 2008.

Cevahir, A., A. Nukada, and S. Matsuoka. Fast Conjugate Gradients with multiple GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, pages 893–903. 2009.

Chand, K. K. Component-based hybrid mesh generation. *International Journal for Numerical Methods in Engineering*, 62:747–773, 2005.

Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2001.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68:1370–1380, 2008.

Chen, Y., T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35:22:1–22:14, 2008.

Chrisochoides, N. Parallel mesh generation. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2005.

Christen, M., O. Schenk, and H. Burkhart. General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform. In *First Workshop on General-Purpose Processing on Graphics Processing Units*. Boston, MA, USA, 2007.

Connolly, D. T. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46:93–100, 1990.

Cooley, J. and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

Cung, V.-D., T. Mautor, P. Michelon, and A. Tavares. A scatter search based approach for the Quadratic Assignment Problem. In *Proceedings of ICEC'97*, pages 165–170. 1997.

Czapiński, M. An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73:1461–1468, 2013.

Czapiński, M. and S. Barnes. Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. *Journal of Parallel and Distributed Computing*, 71:802–811, 2011.

Czapiński, M., C. Thompson, and S. Barnes. Reducing communication overhead in multi-GPU hybrid solver for 2D Laplace's Equation. *International Journal of Parallel Programming*, DOI: 10.1007/s10766-013-0293-2:1–16, 2013.

Davidson, A., Y. Zhang, and J. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 956–965. 2011.

Davis, T. A. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30:196–199, 2004a.

Davis, T. A. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30:165–195, 2004b.

Davis, T. A. and I. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997.

Davis, T. A. and I. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25:1–20, 1999.

Davis, T. A. and W. W. Hager. Modifying a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:606–627, 1999.

Davis, T. A. and W. W. Hager. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 22:997–1013, 2001.

Davis, T. A. and W. W. Hager. Row modifications of a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 26:621–639, 2005.

Davis, T. A. and W. W. Hager. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions on Mathematical Software*, 35:27:1–27:23, 2009.

Davis, T. A. and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1–1:25, 2011.

de Cravalho Jr., S. and S. Rahman. Microarray layout as Quadratic Assignment Problem. In *Proceedings of the German conference on bioinformatics*, pages 11–20. Tübingen, Germany, 2006.

de la Asunción, M., J. M. Mantas, M. J. Castro, and E. Fernández-Nieto. An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems. *Journal of Parallel and Distributed Computing*, 72:1065–1072, 2012.

Demmel, J. Solving the discrete Poisson equation using Jacobi, SOR, Conjugate Gradients, and the FFT. `http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html`, 1996. Last access: 17 June 2013.

Demmel, J. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

Demmel, J. W., S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20:720–755, 1999.

Di, P., H. Wu, J. Xue, F. Wang, and C. Yang. Parallelizing SOR for GPGPUs using alternate loop tiling. *Parallel Computing*, 38:310–328, 2012.

Dickey, J. and J. Hopkins. Campus building arrangement using TOPAZ. *Transportation Science*, 6:59–68, 1972.

Dolbeau, R., S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–5. 2007.

Dongarra, J., I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra on High-Performance Computers*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, 1998.

Dongarra, J. and M. Gates. Freely available software for linear algebra. `http://www.netlib.org/utk/people/JackDongarra/la-sw.html`, 2013. Last access: 11 March 2014.

Drezner, Z. A new genetic algorithm for the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 15:320–330, 2003.

Dubois, P., A. Greenbaum, and G. Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22:257–268, 1979.

Duff, I., A. Erisman, and J. Reid. *Direct methods for sparse matrices*. Monographs on Numerical Analysis. Clarendon Press, 1986.

Duff, I. S. and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equation. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

Duffy, D. *Transform Methods for Solving Partial Differential Equations*. Symbolic & Numeric Computation. Taylor & Francis, second edition, 2010.

Eiselt, H. A. and G. Laporte. A combinatorial optimization problem arising in dartboard design. *The Journal of the Operational Research Society*, 42:113–118, 1991.

Elsen, E., P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227:10148–10161, 2008.

Elshafei, A. N. Hospital layout as a Quadratic Assignment Problem. *Operations Research Quarterly*, 28:167–179, 1977.

Evans, D. J. Parallel S.O.R. iterative methods. *Parallel Computing*, 1:3–18, 1984.

Eymard, R., T. Gallouët, and R. Herbin. The Finite Volume Method. In P. Ciarlet and J. Lions, editors, *Handbook for Numerical Analysis*, pages 715–1022. North Holland, 2000.

Falgout, R. D. An introduction to Algebraic Multigrid. *Computing in Science and Engineering*, 8:24–33, 2006.

Fang, X. G. and G. Havas. On the worst-case complexity of integer Gaussian elimination. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 28–31. ACM, New York, USA, 1997.

Feng, Z. and P. Li. Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 647–654. 2008.

Ferreira, A., E. Kansa, G. Fasshauer, and V. Leitão. *Progress on Meshless Methods*. Computational Methods in Applied Sciences. Springer, 2008.

Ferziger, J. and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, London, 2002.

Fletcher, R. Conjugate Gradient methods for indefinite systems. In G. Watson, editor, *Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Springer, Berlin/Heidelberg, 1976.

Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

Freund, R. W. and N. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

Fries, T. P. and H. G. Matthies. Classification and overview of Meshfree Methods. In *International Conference on Scientific Computing*. 2003.

Frigo, M. and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93:216–231, 2005.

Gambardella, L. M., E. D. Taillard, and M. Dorigo. Ant colonies for the Quadratic Assignment Problem. *Journal of the Operational Research Society*, 50:167–176, 1999.

Gander, M. J. Optimized Schwarz methods. *SIAM Journal on Numerical Analysis*, 44:699–731, 2006.

Gander, M. J., L. Halpern, and F. Magoulès. An optimized Schwarz method with two-sided Robin transmission conditions for the Helmholtz equation. *International Journal for Numerical Methods in Fluids*, 55:163–175, 2007.

Garey, M. R. and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, 1979.

Garland, M., S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, 2008.

Geist, A. *PVM: Parallel Virtual Machine: A users' guide and tutorial for networked parallel computing*. Scientific and engineering computation. MIT Press, 1994.

George, T., V. Saxena, A. Gupta, A. Singh, and A. Choudhury. Multifrontal factorization of sparse SPD matrices on GPUs. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 372–383. 2011.

Giraud, L., R. Guivarch, and J. Stein. A parallel distributed fast 3D poisson solver for Méso-NH. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 1431–1434. Springer-Verlag, 1999.

Giraud, L. and R. Tuminaro. Algebraic domain decomposition preconditioners. Technical Report RT/APO/06/07, ENSEEIHT-IRIT, Toulouse, France, 2006.

Glover, F. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13:533–549, 1986.

Glover, F. A template for Scatter Search and Path Relinking. In *Selected Papers from the Third European Conference on Artificial Evolution*, AE '97, pages 3–54. Springer-Verlag, London, UK, 1998.

Göddeke, D., S. Buijssen, H. Wobker, and S. Turek. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. In *International Conference on High Performance Computing Simulation*, pages 12–21. 2009.

Göddeke, D. and R. Strzodka. Cyclic Reduction tridiagonal solvers on GPUs applied to mixed-precision Multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22:22–32, 2011.

Göddeke, D., R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing*, 33:685–699, 2007a.

Göddeke, D., R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve Multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering*, 4:36–55, 2008.

Göddeke, D., R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22:221–256, 2007b.

Goertzel, G. An algorithm for the evaluation of finite trigonometric series. *The American Mathematical Monthly*, 65:34–35, 1958.

Golub, G. H. and C. F. Van Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, USA, 3rd edition, 1996.

Goodnight, N., C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A Multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111. 2003.

Gould, N. I. M., J. A. Scott, and Y. Hu. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 33:Article No. 10 (32 pages), 2007.

Grcar, J. F. How ordinary elimination became Gaussian elimination. *Historia Mathematica*, 38:163–218, 2011.

Griebel, M. and M. Schweitzer. *Meshfree methods for partial differential equations V*. Lecture notes in computational science and engineering. Springer, 2010.

Gropp, W., E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, USA, 2nd edition, 1999.

Guennebaud, G., B. Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010. Last access: 2 October 2013.

Guo, P. and L. Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 2010 International Conference on Computational and Information Sciences*, pages 1154–1157. 2010.

Gupta, A. and H. Avron. WSMP: Watson Sparse Matrix Package. Part I — direct solution of symmetric systems. Technical Report RC 21886, IBM Research Division, 2000.

Gupta, R. and S. S. Vadhiyar. An efficient MPI_allgather for grids. In *Proceedings of the 16th international symposium on high performance distributed computing*, HPDC '07, pages 169–178. ACM, New York, USA, 2007.

Gustafson, J. L. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

Harris, M. Optimizing parallel reduction in CUDA. `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`, 2008. Last access: 17 January 2013.

Harris, M. How to overlap data transfers in CUDA C/C++. `https://developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc`, 2012. Last access: 29 July 2013.

Harris, M., S. Sengupta, and J. D. Owens. Parallel prefix sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

Hatanaka, M., A. Hori, and Y. Ishikawa. Optimization of MPI persistent communication. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 79–84. ACM, New York, USA, 2013.

Heath, M. T., E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

Heroux, M., R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

Heroux, M. A., R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31:397–423, 2005.

Hestenes, M. R. and E. Stiefel. Methods of Conjugate Gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

Hockney, R. W. A fast direct solution of Poisson's Equation using Fourier analysis. *Journal of the ACM*, 12:95–113, 1965.

Hoefler, T., P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient solver with non-blocking collective operations. *Parallel Computing*, 33:624–633, 2007.

Huang, C.-Y. Recent progress in multiblock hybrid structured and unstructured mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 150:1–24, 1997.

Idelsohn, S. R., E. Oñate, N. Calvo, and F. Del Pin. The meshless finite element method. *International Journal for Numerical Methods in Engineering*, 58:893–912, 2003.

Intel. Math Kernel Library — LINPACK download. `http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download`, 2012. Last access: 9 July 2013.

Iwamura, C., F. S. Costa, I. Sbarski, A. Easton, and N. Li. An efficient Algebraic Multigrid Preconditioned Conjugate Gradient solver. *Computer Methods in Applied Mechanics and Engineering*, 192:2299–2318, 2003.

Jacobsen, D., J. C. Thibault, and I. Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *American Institute of Aeronautics and Astronautics (AIAA) 48th Aerospace Science Meeting Proceedings*. 2010.

James, T., C. Rego, and F. Glover. Multistart Tabu Search and diversification strategies for the Quadratic Assignment Problem. *IEEE Transactions on Systems, Man and Cybernetics — Part A: Systems and Humans*, 39:579–596, 2009.

Janiak, A., W. Janiak, and M. Lichtenstein. Tabu Search on GPU. *Journal of Universal Computer Science*, 14:2416–2427, 2008.

Jeffers, J. and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, Waltham, MA, USA, 2013.

Kaniel, S. Estimates for some computational techniques in linear algebra. *Mathematics of Computation*, 20:369–369, 1966.

Karp, A. H. and H. P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33:539–543, 1990.

Keller, J., C. Kessler, and J. Träff. *Practical PRAM programming*. Wiley Series on Parallel and Distributed Computing. J. Wiley, 2001.

Kim, H.-S., S. Wu, L.-W. Chang, and W.-M. Hwu. A scalable tridiagonal solver for GPUs. In *2011 International Conference on Parallel Processing (ICPP)*, pages 444–453. 2011.

Kincaid, D. and E. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. American Mathematical Society, third edition, 2002.

Kirk, D., W. Hwu, and W. Hwu. *Programming massively parallel processors: a hands-on approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, Burlington, MA, USA, 2010.

Knupp, P. M. and S. Steinberg. *The fundamentals of grid generation*. CRC Press, Boca Raton, FL, USA, 1994.

Koopmans, T. C. and M. J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

Kourtis, K., G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th conference on Computing frontiers*, pages 87–96. 2008.

Krarup, J. and P. M. Pruzan. Computer-aided layout design. *Mathematical Programming Study*, 9:75–94, 1978.

Krawezik, G. P. and G. Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*. 2010.

Lanczos, C. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45:255–282, 1950.

Lanczos, C. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53, 1952.

Langou, J., J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 2006.

Lawlor, O. S. Message passing for GPGPU clusters: cudaMPI. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. 2009.

Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation

of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38:451–460, 2010.

Leonard, B. P. The ULTIMATE conservative difference scheme applied to unsteady one-dimensional advection. *Computer Methods in Applied Mechanics and Engineering*, 88:17–74, 1991.

LeVeque, R. J. *Finite Volume Methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

Li, J., G. Tan, M. Chen, and N. Sun. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. *ACM SIGPLAN Notices*, 48:117–126, 2013.

Li, X. S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31:302–325, 2005.

Liang, Y. *The Use of Parallel Polynomial Preconditioners in the Solution of Systems of Linear Equations*. Ph.D. thesis, University of Ulster, 2005.

Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

Liseĭkin, V. *Grid generation methods*. Springer-Verlag, Germany, 1999.

Loiola, E. M., N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido. A survey for the Quadratic Assignment Problem. *European Journal of Operational Research*, 176:657–690, 2007.

Loisel, S. and D. B. Szyld. On the convergence of Optimized Schwarz Methods by way of matrix analysis. In *Domain Decomposition Methods in Science and Engineering XVIII*, volume 70 of *Lecture Notes in Computational Science and Engineering*, pages 363–370. Springer, Berlin-Heidelberg, 2009.

Luong, T. V., N. Melab, and E.-G. Talbi. Parallel local search on GPU. Technical report, Institut National de Recherche en Informatique et en Automatique, Lille, 2009.

Löhner, R., D. Sharov, H. Luo, and R. Ramamurti. Overlapping unstructured grids. In *39th Aerospace Sciences Meeting and Exhibit*, AIAA-01-0439. 2001.

Martin, R., G. Peters, and J. Wilkinson. Iterative refinement of the solution of a positive definite system of equations. *Numerische Mathematik*, 8:203–216, 1966.

Merz, P. and B. Freisleben. Fitness landscape analysis and memetic algorithms for the Quadratic Assignment Problem. *IEEE Transactions on Evolutionary Computation*, 4:337–352, 2000.

Micikevicius, P. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. 2009.

Moler, C. B. Iterative refinement in floating point. *Journal of the ACM*, 14:316–321, 1967.

Nath, R., S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24:511–515, 2010.

Nickolls, J., I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6:40–53, 2008.

Nukada, A. and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10. ACM, New York, USA, 2009.

NVIDIA. CUDA C programming guide, version 4.0. `http://developer.nvidia.com/cuda-toolkit-40`, 2011. The most recent version (5.5) is available on-line at `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` Last access: 8 April 2014.

NVIDIA. CUDA Basic Linear Algebra Subprograms (CUBLAS). `http://docs.nvidia.com/cuda/cublas/index.html`, 2012a. Last access: 1 July 2013.

NVIDIA. CUDA Fast Fourier Transform (CUFFT) library. `http://docs.nvidia.com/cuda/cufft/index.html`, 2012b. Last access: 1 July 2013.

NVIDIA. CUDA Sparse Matrix Library (CUSPARSE). `https://developer.nvidia.com/cuSPARSE`, 2014. Last access: 5 Feb 2014.

OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, 2008. Last access: 20 September 2013.

Orkisz, J. Finite difference method (part III). In M. Kleiber, editor, *Handbook of Computational Solid Mechanics*, pages 336–432. Springer-Verlag, Berlin, 1998.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80–113, 2007.

Paige, C. C. and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

Patankar, S. V. *Numerical Heat Transfer and Fluid Flow*. Taylor & Francis, 1980.

Pennycook, S. J., S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38:23–29, 2011.

Pepper, D. W. and J. C. Heinrich. *The Finite Element Method: Basic Concepts and Applications*. Taylor & Francis Group, London, 1992.

Pereira, F. H., S. L. L. Verardi, and S. I. Nabeta. A fast algebraic multigrid preconditioned conjugate gradient solver. *Applied Mathematics and Computation*, 179:344–351, 2006.

Perona, P. and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

Pichel, J., D. Singh, and J. Carretero. Reordering algorithms for increasing locality on multicore processors. In *10th IEEE International Conference on High Performance Computing and Communications*, pages 123–130. 2008.

Pickering, M. *An introduction to fast fourier transform methods for partial differential equations, with applications*. Research Studies Press, 1986.

Pollatschek, M., N. Gershoni, and Y. Radday. Optimization of the typewriter keyboard by simulation. *Angewandte Informatik*, 17:438–439, 1976.

Pope, S. B. *Turbulent Flows*. Cambridge Press, 2000.

Press, W., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 3rd edition, 2007.

Püschel, M. and J. M. F. Moura. Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing*, 56:1502–1521, 2007.

Quinn, M. J. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

Reddy, J. N. *An introduction to the Finite Element Method*. McGraw-Hill, 1984.

Reid, J. K. On the method of Conjugate Gradients for the solution of large sparse systems of linear equations. In J. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, Inc., London and New York, 1971.

Rodriguez, M. R., B. Philip, Z. Wang, and M. Berrill. Block-relaxation methods for 3D constant-coefficient stencils on GPUs and multicore CPUs. *CoRR*, abs/1208.1975:1–16, 2012.

Ryoo, S., C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. 2008.

Saad, Y. *Iterative methods for sparse linear systems*. SIAM, second edition, 2003.

Saad, Y. and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 7:856–869, 1986.

Saad, Y. and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123:1–33, 2000.

Sanders, J. and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

Schauer, B. Multicore processors — A necessity. `http://www.csa.com/discoveryguides/multicore/review.pdf`, 2008. Last access: 17 August 2014.

Schenk, O. and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20:475–487, 2004.

Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPH-ICS symposium on graphics hardware*, pages 97–106. 2007.

Serafini, D. B., P. McCorquodale, and P. Colella. Advanced 3D poisson solvers and particle-in-cell methods for accelerator modeling. *Journal of Physics: Conference Series*, 16:481–485, 2005.

Shanley, T., J. Winkles, and I. MindShare. *InfiniBand Network Architecture*. PC system architecture series. Addison-Wesley, 2003.

Shet, A., P. Sadayappan, D. Bernholdt, J. Nieplocha, and V. Tipparaju. A framework for characterizing overlap of communication and computation in parallel applications. *Cluster Computing*, 11:75–90, 2008.

Shewchuk, J. R. An introduction to the Conjugate Gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, USA, 1994.

Sonneveld, P. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10:36–52, 1989.

Steinberg, L. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.

Stock, F. and A. Koch. A fast GPU implementation for solving sparse ill-posed linear equation systems. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, pages 457–466. 2010.

Stone, H. S. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20:27–38, 1973.

Stratton, J., S. Stone, and W.-m. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In J. Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin/Heidelberg, 2008.

Stützle, T. Iterated local search for the Quadratic Assignment Problem. *European Journal of Operational Research*, 174:1519–1539, 2006.

Stützle, T. and M. Dorigo. ACO algorithms for the Quadratic Assignment Problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 33–50. McGraw-Hill, 1999.

Taillard, E. Robust taboo search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

Temam, R. *Navier-Stokes Equations: Theory and Numerical Analysis*. AMS Chelsea Publishing. American Mathematical Society, 2001.

Thakur, R. and W. Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35:608–617, 2009.

Thomas, L. *Elliptic Problems in Linear Differential Equations over a Network*. Watson Sci. Comput. Lab. Rept., Columbia University, New York, 1949.

Tomov, S., J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36:232–240, 2010a.

Tomov, S., R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. 2010b.

van der Sluis, A. and H. A. van der Vorst. The rate of convergence of Conjugate Gradients. *Numerische Mathematik*, 48:543–560, 1986.

van der Vorst, H. A. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13:631–644, 1992.

Van Loan, C. *Computational Frameworks for the Fast Fourier Transform.* Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, 1992.

Varga, R. *Matrix Iterative Analysis.* Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 1999.

Versteeg, H. and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method.* Pearson Education Ltd., 2007.

Volkov, V. and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, University of California, Berkeley, 2008.

Whaley, R. C. and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35:101–121, 2005.

Whaley, R. C., A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

White, J., III and J. Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *International Parallel and Distributed Processing Symposium*, pages 59–67. 2011.

Wilkinson, J. *The algebraic eigenvalue problem.* Monographs on numerical analysis. Clarendon Press, 1965.

Willcock, J. and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 307–316. 2006.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35:178–194, 2009.

Wilmott, P., S. Howison, and J. Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction.* Cambridge University Press, 1995.

Wozniak, M., T. Olas, and R. Wyrzykowski. Parallel implementation of Conjugate Gradient method on graphics processors. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, pages 125–135. 2010.

Wu, J. and J. JaJa. Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs. In *Innovative Parallel Computing*, pages 1–12. 2012.

Wu, J., J. JaJa, and E. Balaras. An optimized FFT-based direct Poisson solver on CUDA GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25:550–559, 2014.

Yan, Y., M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 887–899. Springer Berlin Heidelberg, 2009.

Yang, C.-T., C.-L. Huang, and C.-F. Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182:266–269, 2011.

Yang, S. and M. K. Gobbert. The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions. *Applied Mathematics Letters*, 22:325–331, 2009.

Young, D. M. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type.* Ph.D. thesis, Harvard University, 1950.

Young, D. M. *Iterative solution of large linear systems.* Academic Press, New York, 1971.

Yu, C. D., W. Wang, and D. Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37:759–770, 2011.

Zhang, Y., J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. *ACM SIGPLAN Notices*, 45:127–136, 2010.

Zhu, W., J. Curry, and A. Marquez. SIMD tabu search for the Quadratic Assignment Problem with graphics hardware acceleration. *International Journal of Production Research*, 48:1035–1047, 2010.

Zienkiewicz, O., R. Taylor, R. Taylor, and J. Zhu. *The Finite Element Method: Its Basis and Fundamentals*. Elsevier Butterworth-Heinemann, 6th edition, 2005.

Appendix

Appendix A

CUDA Platform Overview

This short overview of the CUDA platform is based on the CUDA C Programming Guide (NVIDIA, 2011). For further information refer to this publication — the most recent version is available on-line at `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

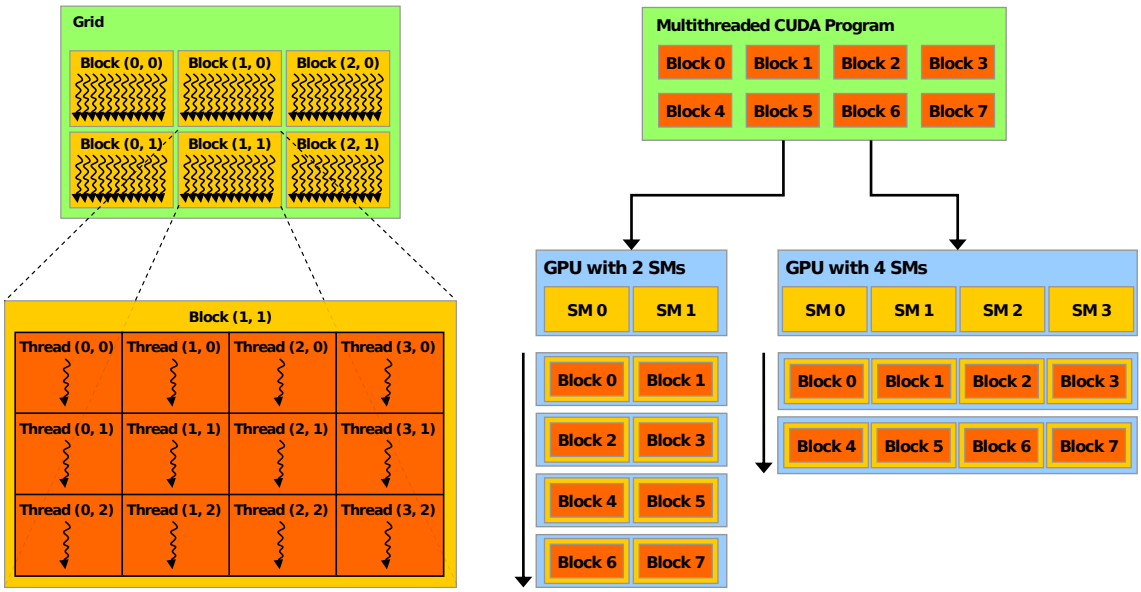
Successive generations of graphics hardware differ significantly in terms of computational capabilities and supported features. These are defined by *Compute Capability* (CC) of the GPU, e.g. the oldest GPUs had no support for double precision arithmetic, which was introduced in CC 1.3 but was over ten times slower than operations in single precision until some devices with CC 2.0. For comparison of different Compute Capabilities refer to the programming guide (NVIDIA, 2011, Appendix F).

The CUDA platform offers abstractions hiding most low-level GPU-related issues in three main areas: thread hierarchy, memory management, and synchronisation. These abstractions are described in the following sections.

Thread Hierarchy

To perform any computation on the GPU, the programmer must define a C function called the *kernel*. This function is then run using a specified number of lightweight threads. Threads are grouped in *blocks*, and blocks form a *grid* (Figure A.1 A).

This hierarchy is introduced to reflect the hardware organisation of the GPU. While threads within a block are able to communicate through shared memory, blocks are



(A) Treads are grouped into blocks, blocks form a computing grid.

(B) Blocks can be processed independently on separate streaming multi-processors.

FIGURE A.1: Thread hierarchy in the CUDA programming model (from NVIDIA, 2011).

meant to run independently on one of the several *streaming multi-processors* (SM) on the GPU (Figure A.1B). Multiple blocks may run concurrently on a single SM, provided that sufficient resources are available (shared memory, registers). However, there is no mechanism to synchronise threads in different blocks — communication between the blocks is impractical.

On an SM, threads are executed in groups of 32 threads (*warps*) in the SIMD model — threads within a warp are executed in a *lock-step*, i.e. they perform the same operation concurrently. In consequence, the performance may be noticeably reduced in the presence of control instructions (`if`, `switch`) that cause threads within a warp to follow different execution paths. The notion of warps is also important to understand how the global and shared memory transactions are executed.

Memory Hierarchy

Several memory types are available on the GPU — they differ in size, visibility, access time, and whether they are cached and writable. Below is the description of

each memory type, its capabilities, and purpose.

Global memory is used to store input and output data and for CPU-GPU communication. It is accessible from the GPU (directly) and the CPU (through the CUDA API). It is the largest memory available on the GPU (a few GB), however it is typically much smaller than CPU memory. Accessing global memory from the GPU is connected with very high latency (typically 300–600 GPU cycles) and relatively low bandwidth (10s or 100s of GB/s), and in consequence, often becomes a performance limiting factor. Cards with Compute Capability 2.0 and higher are equipped with L1 (per SM) and L2 (per GPU) global memory cache providing a significant improvement over the first generation devices.

Shared memory is a fast on-chip memory available for both reading and writing but only visible by threads within the same block. Its main purpose is to allow communication between the threads in a block, however it may be also used to act as a cache. Shared memory is typically very small — 16 KB or 48 KB per streaming multi-processor is available. If a CUDA kernel uses a lot of shared memory, then the number of blocks resident on an SM (*device occupancy*) is limited. High occupancy helps to hide the latency connected with global memory access and is essential for good performance of memory-bound kernels.

Registers form the fastest on-chip memory used for automatic variables of each thread. The number of registers on an SM is limited (typically 10s of 32-bit registers per thread), therefore if a CUDA kernel uses many registers, then the occupancy may be reduced. If each thread in a kernel stores large automatic variables (e.g. arrays or complex structures) that would not fit in the registers, the compiler may move them to so called *local memory*, which is implemented in global memory, and in consequence it is a few orders of magnitude slower.

Constant memory is accessible in the same way as global memory, however it is cached and read-only from the GPU. It is relatively small (64 KB per GPU, 8 KB physical cache), therefore on devices with global memory caching (CC 2.0 and higher) there is no need to use it explicitly. Even on the most recent GPUs, constant memory is used for kernel call parameters marshaling.

Texture memory was widely used to provide global memory caching on the first generation graphics cards. From the GPU, it is read-only and has to be accessed via the texture fetch API. Texture cache is optimised for data reads with 2D spatial locality, e.g. stencil operators on 2D grids.

Surface memory was introduced on devices with Compute Capability 2.0. It offers functionality similar to texture memory, however in addition it supports writing from the GPU kernels.

As mentioned above, accessing global memory is connected with very high latency. To alleviate this limitation, global memory requests for threads within a single warp are *coalesced* into fewer large transactions. The conditions necessary for the maximum level of coalescing vary between devices with different Compute Capability, however typically it is required that all threads access data within properly aligned 32-, 64-, or 128-byte block.

Similar to global memory, threads in a single warp can access shared memory in parallel. This is possible since data in shared memory is divided between 16 or 32 memory banks, depending on Compute Capability. However, to achieve the best performance each thread must access data in a different bank, or multiple threads must read from the same address (*broadcast* operation). This is easily achieved if threads read/write consecutive addresses, but if access is strided, then bank conflicts may occur leading to serialised memory transactions.

Maximising the level of global memory coalescing and avoiding shared memory bank conflicts is crucial to achieve high performance in many applications, especially if the kernel is memory-bound. Often the data layout must be adjusted to enable these optimisations, e.g. deciding whether to follow an array-of-structures or structure-of-arrays approach.

Memory transfers between the CPU and the GPU are performed via a PCI Express connection, which offers several GB/s of bandwidth. The CPU-GPU communication speed can be improved by using *page-locked* memory on the CPU. However, this mechanism should be used carefully as it reduces the amount of physical memory available to the operating system — page-locked memory cannot be used to store virtual memory pages.

Synchronisation

CUDA provides an efficient mechanism to synchronise threads within a block. The `__syncthreads()` command is a simple barrier operation that forces threads to wait until all the other threads reach that point in code. Thread synchronisation mechanism is essential for inter-thread communication using shared memory — if threads do not synchronise when writing to shared memory, then a race condition may occur and lead to undefined and difficult to debug behaviour.

All CUDA kernels are executed asynchronously with respect to the CPU — the control is returned immediately after the kernel is dispatched for execution. This behaviour allows the CPU to execute code in parallel to the GPU computation. Furthermore, processing on the GPU can be controlled using multiple CUDA streams — instructions issued to different streams are treated as independent and can possibly run concurrently on the GPU (this includes memory transfers). On the other hand, GPU operations are serialised within each CUDA stream and they are performed in first-in, first-out (FIFO) order. The CUDA API provides several functions to synchronise the CPU with the GPU: the entire device, a single stream, or even a specific event within a CUDA stream.

By default, data transfers between the CPU and the GPU are blocking and force synchronisation with the GPU. To minimise the time during which the GPU waits idle for input data, cards with Compute Capability 1.1 and higher can overlap CUDA kernel execution with transferring data between the CPU and the GPU (one-way only). Overlapping is only possible if the following conditions are met: memory on the CPU must be page-locked, the asynchronous memory transfer API must be used, and computation and communication must be run in different CUDA streams. On devices with Compute Capability 2.0 and higher, it is possible to overlap computation with two-way memory transfers providing the GPU is equipped with two asynchronous copy engines (typically only present on HPC-grade cards).

Appendix B

Hardware Specifications

Table B.1: Specifications of CPUs used in experiments.

	Intel Core i7-980x	Intel Xeon E5462
Clock speed	3,333 MHz	2,800 MHz
Front-Side Bus speed	3,200 MHz	1,600 MHz
Cores count	1×6	2×4
L1 cache	6×32 KB	4×32 KB
L2 cache	6×256 KB	—
L3 cache	12 MB	2×6 MB

Table B.2: Specifications of GPUs used in experiments.

	Tesla C1060	Tesla C2050	GTX 480
Compute capability	1.3	2.0	2.0
Cores	240 (30×8)	448 (14×32)	480 (15×32)
GPU clock rate	1,296 MHz	1,147 MHz	1,401 MHz
Peak performance (float)	622 GFLOPS	1,027 GFLOPS	1,345 GFLOPS
Memory size	4 GB	3 GB	1.5 GB
Memory clock	800 MHz	1,500 MHz	1,848 MHz
Memory bus	512-bit	384-bit	384-bit
Peak performance	102 GB/s	144 GB/s	177 GB/s
PCI-Express version	1.0	2.0	2.0
Peak performance	4 GB/s	8 GB/s	8 GB/s
Asynchronous copy engines	1	2	1

Appendix C

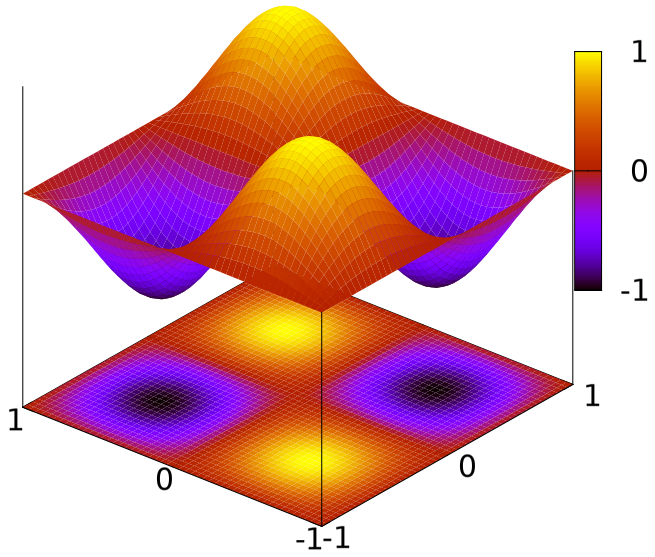
Model Problems Used in Experiments

Table C.1: Sparse matrices from University of Florida Collection (Davis and Hu, 2011).

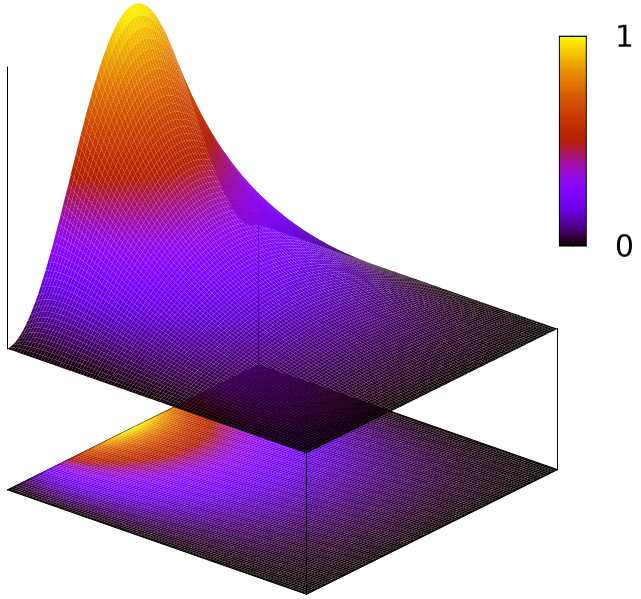
Matrix name	Size (n)	Non-zero elements (nnz)	SPD?
`bbmat`	38,744	1,771,722	no
`GT01R`	7,980	430,909	no
`lung2`	109,460	492,564	no
`shyy161`	76,480	329,762	no
`crankseg_1`	52,804	10,614,210	yes
`Geo_1438`	1,437,960	63,156,690	yes
`hood`	220,542	9,895,422	yes
`parabolic_fem`	525,825	3,674,625	yes
`StocF-1465`	1,465,137	21,005,389	yes
`thermal1`	82,654	574,458	yes

Table C.2: Matrices from FDM discretisation of 2D and 3D Poisson's Equations.

Grid size	Matrix size (n)	Non-zero elements (nnz)	Stencil
64×64	4,096	20,224	5-point
256×256	65,536	326,656	5-point
512×512	262,144	1,308,672	5-point
$1{,}024 \times 1{,}024$	1,048,576	5,238,784	5-point
$2{,}048 \times 2{,}048$	4,194,304	20,963,328	5-point
$16 \times 16 \times 16$	4,096	27,136	7-point
$64 \times 64 \times 64$	262,144	1,810,432	7-point



(A) Domain: $\Omega = \{(x, y) : -1 < x,\ y < 1\}$
Boundary conditions: $\phi = 0$ on $\partial\Omega$ (homogeneous)
Source term: $f = -2\pi^2 \sin(\pi x) \sin(\pi y)$ in Ω
Analytic solution: $\phi = \sin(\pi x) \sin(\pi y)$



(B) Domain: $\Omega = \{(x, y) : 0 < x,\ y < 1\}$
Boundary: $\phi(x, y) = \begin{cases} \sin^2(\pi y) & \text{if } x = 0 \\ 0 & \text{if } x = 1,\ y = 0, \text{ or } y = 1 \end{cases}$ on $\partial\Omega$
Source term: $f \equiv 0$ in Ω (2D Laplace's Equation)

FIGURE C.1: Solutions to the 2D Poisson's Equation: $\nabla^2 \phi = f$.

Appendix D

List of Publications

Research Papers

- Czapiński, M., C. Thompson and S. Barnes. Reducing communication overhead in multi-GPU Hybrid Solver for 2D Laplace's Equation. *International Journal of Parallel Programming*, `http://dx.doi.org/10.1007/s10766-013-0293-2`, 2013
- Czapiński, M. An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73 (11) 1461–1468. 2013.
- Czapiński, M. and S. Barnes. Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. *Journal of Parallel and Distributed Computing*, 71 (6):802–811. 2011.

Conference Talks

- Czapiński, M. An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem on CUDA platform. 4^{th}*International Many-core and Reconfigurable Supercomputing Conference*, MRSC 2011. Bristol, UK. $12^{\text{th}}$of April 2011.